

C/C++ 学习指南（语法篇）

作者：邵发

官网：<http://afanihao.cn>

QQ 群：417024631



“让编程变得简单！”

相关资源：

- (1) 配套 90 集视频教程，可独立使用，也可以结合使用
- (2) 配套在线题库，配备答案和讲解
- (3) 提示所有示例源码下载
- (4) 配套后续 C/C++ 开发相关的全套进阶教程（Linux 开发/QT 界面开发/多线程与网络编程等等）

1. 开始学习 C/C++	5
1.1 开发平台	5
1.2 第一个程序	5
1.3 代码与程序	7
1.4 C 语言和 C++ 语言	7
2. 控制台输入与输出	8
2.1 控制台输出	8
2.2 控制台输入	13
2.3 注释	16
2.4 空白	17
2.5 常见问题	18
2.6 综合例题	19
3. 变量与常量	20
3.1 变量	20
3.2 整型变量	23
3.3 浮点型变量	25
3.4 数的进制表示	26
3.5 变量与内存	28
3.6 CONST 常量	31
3.7 *字面常量	32
3.8 常用类型的范围	33
4. 数组	35
4.1 引例	35
4.2 数组的定义	35
4.3 数组的基本方法	36
4.4 数组的内存视图	38
4.5 常见问题	39
4.6 数组的使用实例	40
4.7 多维数组	41

5. 字符与字符数组	43
5.1 字符是什么	43
5.2 字符的表示	43
5.3 字符的显示	44
5.4 字符常量	44
5.5 字符数组	45
5.6 转义字符	48
6. 表达式与操作符	50
6.1 算术表达式	50
6.2 赋值表达式	51
6.3 关系表达式	53
6.4 条件表达式	54
6.5 逻辑表达式	55
6.6 逗号表达式	56
6.7 自增/自减操作符	57
6.8 *位操作符	58
6.9 类型的转换与提升	64
6.10 优先级与结合顺序	65
7. 语句	67
7.1 什么叫语句	67
7.2 IF 语句	69
7.3 SWITCH 语句	74
7.4 FOR 语句	79
7.5 WHILE 语句	83
7.6 DO...WHILE 语句	86
7.7 综合例题 1	88
7.8 综合例题 2	89
8. 函数	91
8.1 引例	91

8.2	初步认识函数.....	91
8.3	函数的定义.....	93
8.4	函数的调用.....	97
8.5	全局变量和局部变量.....	101
8.6	变量的作用域与生命期.....	102
8.7	变量名重名问题.....	103
8.8	函数声明与函数定义.....	105
8.9	MAIN 函数.....	109
8.10	参数的隐式转换.....	109
8.11	*函数名重载.....	110
8.12	*重载函数的匹配.....	111
8.13	*参数的默认值.....	113
8.14	*内联函数.....	115
8.15	*函数的递归调用.....	116
9.	附录用 VC2008 创建项目.....	119
9.1	新建 C++ 项目.....	119
9.2	查看项目文件夹.....	121
9.3	添加文件到项目.....	121
9.4	删除项目中的文件.....	124
9.5	编译项目.....	125
9.6	运行程序.....	126

1. 开始学习 C/C++

计算机上可以运行多种程序，比如 Word 程序，可用于编辑和保存文档。比如 QQ 程序，可用于网络聊天。比如浏览器程序，可用于在线浏览网站内容。那么这些程序本身又是怎么做出来的呢？

C/C++ 语言就是一门用于制作程序的技术。

1.1 开发平台

在开始学习 C/C++ 语言之前，需要先安装好相关的开发工具，或称开发平台。本书推荐安装 VS2008 或 VS2010, VS2012，其中 VS 是 Visual Studio 的简称，它是微软公司推出的开发平台。它支持多种编程语法，C++ 只是其中之一，我们将 VS 里面的 C++ 环境也称为 VC。

理论上，读者也可以自选其他的编译器或开发平台，例如 gcc, Eclipse, Dev-C++, C-Free 等开发工具。但对于初学者来说，为了避免不必要的困扰，还是建议使用 VS 平台。

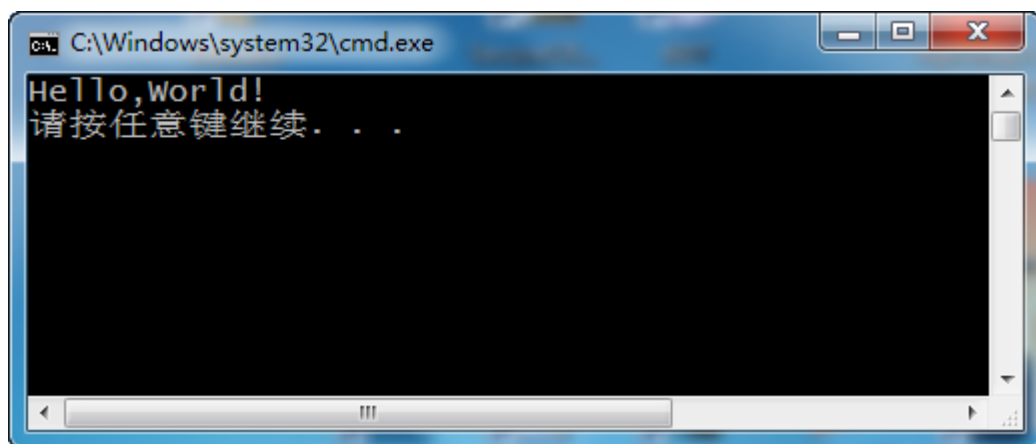
1.2 第一个程序

跟大多数编程语言的教程一样，我们的要创建的第一个程序也是 HelloWorld 程序。对照本书附录《用 VC2008 创建项目》，来建立我们的第一程序。

```
////////// CH01_A1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello,World!\n");  
    return 0;  
}
```

在本教程里，对每一个示例进行编号，比如 CH01_A1 就是这个例子的编号。读者可以官网上下载到所有的示例源码。

我们按照附录里的说明，按 F7 编译这个程序，然后按 CTRL+F5 运行程序，将会弹出来一个黑色的窗口。我们把这个窗口称之为控制台窗口，简称控制台。如下图所示。



1.2.1 代码解析

这个 HelloWorld 程序代码虽然只有 6 行，但是它的语法细节覆盖前 18 章的内容。因此，对初学者来，是没有必要在第一节就掌握它的所有含义的。

对于本书的读者，在前面 7 章的学习中，大家只要把代码框架照抄下来即可。至于其中的语法细节，会循序渐进地给大家介绍。本书示例使用的代码框架为：

```
#include <stdio.h>

int main()
{

    return 0;
}
```

也就是，这几行代码框架是大家暂时不需要关心的，直接照抄就可了。在每一章，我们都会学习一些新的语法，我们会把相关的练习代码填在 `return 0` 这一行的前面。

1.2.2 在 Windows XP 环境下的问题

如果你的操作系统还是以前的 Windows XP，那么可能发现在运行程序的时候，还没来得及查看内容，窗口自己就关闭了。这个问题的解决办法是，使用以下的代码框架：

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    printf("Hello,World!\n");
}
```

```
    system("PAUSE");  
    return 0;  
}
```

1.2.3 常见的问题

在创建第一个程序时，大家最常见的问题：

(1) 标点符号

在示例程序中，标点符号均为英文标点（半角）。而初学者经常会把它写成中文的标点（全角），这样在编译时编译器会报错。

例如，大括号内的每一行末尾有个分号，字符串"Hello,World"是用双引号包围的，等等。

(2) 大小写

对于前面所列的代码框架，每一个字母的大小写是严格区分的，不要把 `main` 写成 `MAIN`。

1.3 代码与程序

我们在 `main.cpp` 文件中写的文本，称为代码（Code）。代码在经过 VC 平台的编译处理之后，生成的 `exe` 文件，称为程序（Program）。程序是交给用户来用的，而代码则是程序员自己保留的。

对于程序员来说，代码是核心价值，是智慧和心血的结晶。程序文件丢了，可以花一分钟重新生成一份；代码如果丢了，那就只能重写了。

1.4 C 语言和 C++ 语言

本书一本 C 语言和 C++ 语言通用的教程。实际上，C 语言只是 C++ 语言前身，对应了本书的第 1-19 章。C++ 语言是在 C 语言的基础上，一方面修改了 C 语言原有的缺陷，另一方面又新增了一些新的语法。当您学全整个教程，就既掌握 C 语言又掌握 C++ 语言了。

2. 控制台输入与输出

控制台是初学者经常要面对的一个界面，它可以输出显示一些字符，也可用键盘输入一些字符。我们今后的练习都是通过控制台来完成的。在这一章，就是初步学习一下，如何向控制台输出显示一些数据，以下如何从控制台接收用户的输入。

2.1 控制台输出

我们使用 `printf` 操作，来向控制台输出数据。其中，`print` 的意思是打印输出，`f` 代表 `format`，整体上就是格式化打印输出的意思。

在示例 CH02_A1 中，我们使用 `printf` 操作输出了两个文本，代码如下所示。

////////// 例 CH02_A1 //////////

```
#include <stdio.h>

int main()
{
    printf("i am shaofa \n");
    printf("我是谁谁谁 \n");
    return 0;
}
```

使用 `printf` 既可以输出英文文本，也可以输出中文文本。由于中文问题在计算机领域是一个复杂的问题，所以作为初学者来说，建议使用英文或拼音来练习。记住：我们的重点是学习它的语法原理，不一定要用中文的。

`printf` 操作的要素包含以下几点：

- ① 小括号：小括号内的东西称为参数列表
- ② 双引号：双引号里的文本就是要输出到控制台内的文本
- ③ `\n`：这个表示换行，具体意义先不用理会，直接照抄即可。

另外，需要对初学者再次强调：

- ① 代码框架照抄即可

在本书示例中，我们总是使用下面的代码框架，至少每一行究竟是什么含义，初学者先不用理会。（涉及了第 1 章~18 章的语句，需逐步介绍）

```
#include <stdio.h>

int main()
{
```



```
    return 0;
}
```

② 读者只需要关心书中讲了什么，而不必关心书中没讲的东西

所有的语法，都是以循序渐进地方式讲明；对于没有讲解地部分，照抄示例代码即可。

2.1.1 输出整数

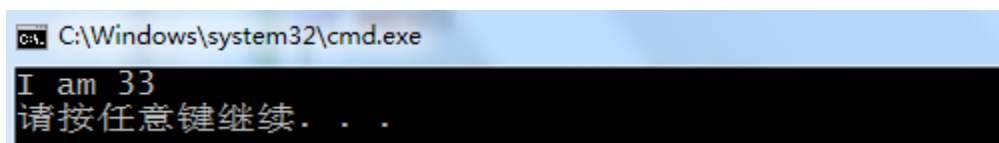
使用 `printf` 可以输出一个整数。下面的例子中，使用 `printf` 输出整数：

////////// 例 CH02_A2 //////////

```
#include <stdio.h>

int main()
{
    printf("I am %d \n" , 33);
    return 0;
}
```

编译这个代码，运行程序，输出显示为：



比较 `printf` 中双引号中的文本，和实际输出的文本，可以发现，在实际的输出结果中对 `%d` 进行了一个替换动作：将 `%d` 替换为了 33。

此例中 `printf` 使用的语法的要点：

- ① 小括号内的参数列表以逗号分开，表示有 2 个参数
- ② 第一个参数: "I am %d \n", 字符串，须以双引号包围
- ③ 第二个参数: 33, 数字

2.1.2 用变量表示整数

我们也可以用下面的方式来输出整数。

////////// 例 CH02_A3 //////////

```
#include <stdio.h>

int main()
{
```

```
int age = 33;

printf("I am %d \n", age);

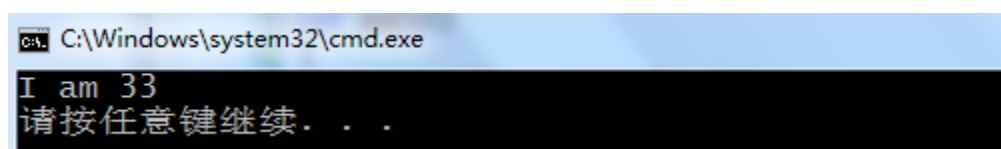
return 0;

}
```

注意：大括号内的每一行的末尾，都须有一个分号作为结束。

在例 CH02_A3，定义了一个变量 `age`，指定它的值为 33。它的类型为 `int`，代表 `integer`（整数）的意思。

我们可以用这样的句子 `int age = 33;`来定义一个变量，然后在 `printf` 里打印输出这个变量。同样的，`printf` 操作会把 `%d` 替换为 `age` 的值。最终输出结果和前一例相同。



也可以在同一行内打印两个整数，例如，

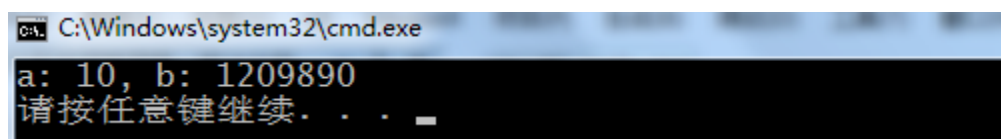
////////// 例 CH02_A4 //////////

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 1209890;
    printf("a: %d, b: %d \n", a, b);
    return 0;
}
```

注意，小括号内有 3 个参数，依次用逗号隔开。

原理仍然一样，`printf` 操作会把 `%d` 替换成后面的数值。此例的运行结果输出为：



2.1.3 指定显示宽度

例如, 当我们输出数字 9, 默认它只占一位宽度, 但我们可以使用%4d 来控制它显示为四位宽度。同理, %5d 表示显示为五位宽度, %8d 表示显示为八位宽度。

下面, 我们通过两个例子, 来比较一下%d 来%4d 的显示效果。

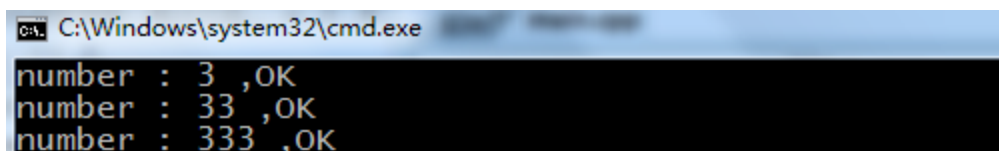
例 CH02_A5 中使用了%d 来显示数字,

////////// 例 CH02_A5 //////////

```
#include <stdio.h>

int main()
{
    printf("number : %d ,OK\n", 3);
    printf("number : %d ,OK\n", 33);
    printf("number : %d ,OK\n", 333);
    return 0;
}
```

运行结果显示为:

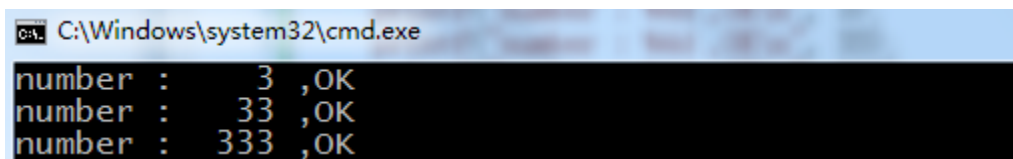


例 CH02_A6 中使用了%4d 来显示数字,

////////// 例 CH02_A6 //////////


```
#include <stdio.h>

int main()
{
    printf("number : %4d ,OK\n", 3);
    printf("number : %4d ,OK\n", 33);
    printf("number : %4d ,OK\n", 333);
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
number : 3 ,OK
number : 33 ,OK
number : 333 ,OK
```

我们还可以使用%04d 来控制显示宽度，表示当位数不足 4 位时，前面填 0 显示。显示效果如下：



```
C:\Windows\system32\cmd.exe
number : 0003 ,OK
number : 0033 ,OK
number : 0333 ,OK
```

2.1.4 输出小数

可以使用 printf 操作来输出小数，使用的格式符为%lf （lf 代表 long float-point）。

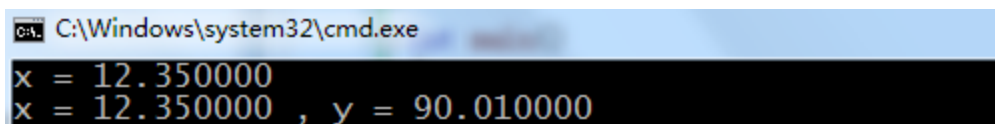
在示例 CH02_B1 中，第一行 printf 输出一个小数，第二行输出了两个小数。代码如下：

////////// 例 CH02_B1 //////////

```
#include <stdio.h>

int main()
{
    printf("x = %lf\n", 12.35);
    printf("x = %lf , y = %lf\n", 12.35, 90.01);
    return 0;
}
```

编译代码，运行程序，结果显示为：



```
C:\Windows\system32\cmd.exe
x = 12.350000
x = 12.350000 , y = 90.010000
```

可见，对于小数来说，printf 就是把%lf 替换为后面的小数的值。

2.1.5 用变量表示小数

我们也可以下面的方式，用一个变量来表示小数。

////////// 例 CH02_B2 //////////

```
#include <stdio.h>

int main()
{
    double a = 12.35;
    printf("x = %lf \n", a);
    return 0;
}
```

在例 CH02_B2 中, 定义了一个变量 a, 其类型为 double, 表示小数。

2.1.6 指定小数点后的位置

我们可以控制小数点后的位数显示。例如, %.2lf, 表示在显示输出的时候, 小数点后面只保留 2 位, 后面的值四舍五入。例如,

////////// 例 CH02_B3 //////////

```
#include <stdio.h>

int main()
{
    double a = 12.35719987;
    printf("x = %.2lf \n", a);
    return 0;
}
```

输出显示为:

A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\system32\cmd.exe'. The command prompt displays the output 'x = 12.36'.

2.2 控制台输入

可以 scanf 操作, 让用户从控制台输入一个整数或小数。其中, scan 表示输入扫描, f 表示 format, 表示接收输入并格式化数据的意思。

2.2.1 输入整数

下面的例子中, 使用 scanf 操作来接收一个整数输入。

////////// 例 CH02_C1 //////////

```
#include <stdio.h>
```

```
int main()
{
    int a = 0;
    scanf("%d", &a);
    printf("Got: %d\n", a);
    return 0;
}
```

在例 CH02_C1 中，首先定义了一个 `int` 类型的变量 `a`，然后调用 `scanf` 操作，保存用户的输入。

编译代码，运行这个程序。在光标闪烁的时候，表示此时用户可以输入一个数（如 123）。用户输入一个整数，按回车结束输入。整个显示为：



这表明，`scanf` 可以接收用户的输入，转化成数值保存在变量 `a` 中。

此时，我们应该注意一下 `scanf` 行的写法：

- ① 小括号内的是参数列表，以逗号分隔
- ② 第一个参数是一个字符串，以双引号包围
- ③ 第二个参数，记得要在 `a` 前面加一个 `&` 号
- ④ 字符串内不要加多余的空格、或其他字母和标点，不是无法接收输入。

初学者往往因为在双引号内加了多余的字符，而得不到正确的结果。例如，如果写成下面这样，是得不到数据的：

```
scanf("%d\n", &a); // 错！双引号内不要加\n
```

2.2.2 输入小数

输入小数的过程也很类似，要用一个 `double` 型变量来存放小数，并使用格式符 `%lf`。下面的例子展示了小数的输入。

```
////////// 例 CH02_C2 //////////
#include <stdio.h>
```

```
int main()
{
    double f = 0;
    scanf("%lf", &f);
    printf("Got: %lf\n", f);
    return 0;
}
```

编译代码并运行程序，在光标闪烁时，输入一个数 12.345，按回车结束输入。程序显示为：



```
C:\Windows\system32\cmd.exe
12.345
Got: 12.345000
```

2.2.3 一次输入多个数

当需要输入多个数时，有两种方法。本教程推荐第（1）种方法，因为它准确可靠，初学者易于掌握、不容易出错。

第（1）种方法：

可以一次接收一个数：每输入一个数，按一次回车。例如，

//////////例 CH02_C3 //////////

```
#include <stdio.h>

int main()
{
    int a;
    double x;
    scanf("%d", &a);
    scanf("%lf", &x);
    printf("Got: %d, %lf\n", a, x);
    return 0;
}
```

运行程序，输入 12，按回车，输入 123.456，按回车。程序显示为：



```
C:\Windows\system32\cmd.exe
12
123.456
Got: 12, 123.456000
```

第(2)种方法:

也可以在一行内输入多个数据, 每个数据以逗号分隔, 然后按回车。例如,

////////// 例 CH02_C4 //////////

```
#include <stdio.h>

int main()
{
    int a;

    double x;

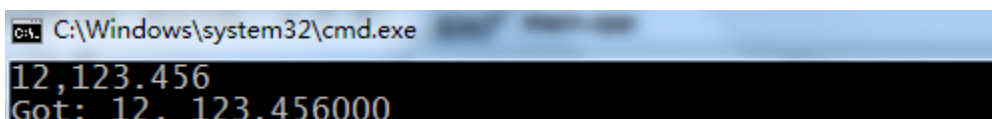
    scanf("%d,%lf", &a, &x);

    printf("Got: %d, %lf\n", a, x);

    return 0;
}
```

在例 CH02_C4 中, 一行 `scanf` 便可以接入两个数, 注意双引号内的字符串 `%d` 和 `%lf` 是以逗号分开的。

运行此程序, 一次性地输入 2 个数, 并以英文逗号隔开, 按回车。显示如下:



```
C:\Windows\system32\cmd.exe
12,123.456
Got: 12, 123.456000
```

2.3 注释

可以在代码里加一点注释性的文字, 它们不影响原代码的运行。注释的作用, 是为了让程序员在阅读起来更清晰, 相当于备注的作用。

在 C/C++ 里, 有两种注释语法。一种以 `//` 开头, 表示单行注释语句。另一种以 `/* */` 包围, 表示任何行数的注释。

第一种注释方法(单行注释语法): 如果你只想写一行注释, 则可以用 `//` 开头, 编译器把该行里面 `//` 后面的所有文字视为注释。

////////// 例 CH02_D1 //////////


```
#include <stdio.h>

// 我的第一个测试

int main()
{
    double a; // 定义 1 个变量

    scanf("%lf", &a); // 接收输入

    printf("Got: %lf\n", a); // 输出显示这个数

    return 0;
}
```

第二种注释方法（多行注释语法）：如果注释文字多于一行，可以用 /* 和 */ 包围起来，编译器将中间的文字视为注释。

```
//////////例 CH02_D2 //////////

#include <stdio.h>

/* 我的第一个测试: 验证 printf 的功能
   它成功了!
*/

int main()
{
    double a = 2.0123;

    printf("%.3lf\n", a);

    return 0;
}
```

2.4 空白

在 C/C++ 代码里，为了美观和可读性，可以在代码里适当一些空白，并不会对程序的产生任何影响。在这里，空白包括空格、制表符和换行。

例如，

```
int a=10;
```

我们可以多点空白，不会对结果产生任何影响。

```
int    a    =    10    ;
```

我们甚至可以把它写在多行里面，在语法上没有影响，结果还是一样，只是代码读起来就比较难看了。例如，

```
////////// 例 CH02_E1 //////////
```

```
#include <stdio.h>

int main()
{
    int a

        = 10;

    printf("Got: %d\n",

        a);

    return 0;
}
```

这样的代码书写方式，虽然在语法上是正确的，而且结果也是正确的。但我们很不提倡，因为它太难以阅读性，降低了代码的可读性。

2.5 常见问题

(1) 格式和类型要对应

在使用 `printf` 和 `scanf` 时，格式符要和类型对应起来。

例如，下面的代码在运行时会有问题：

```
int a = 1;

double b = 2.2;

printf("%lf, %d ", a, b);
```

应该是

```
printf("%d, %lf ", a, b);
```

因为 `%d` 是用于打印整数，而 `%lf` 是用于打印小数的，类型要对应起来。

(2) `scanf` 内加了多余的字符

例如，

```
int a;

scanf("%d\n", &a); // 错了，不要加空格，也不要加\n
```

2.6 综合例题

写一个程序，让用户输入两个数（允许是小数），然后将这两个数的乘积输出。

//////////例 CH02_F1 //////////

```
#include <stdio.h>

int main()
{
    double a = 0; // 第一个数
    double b = 0; // 第二个数

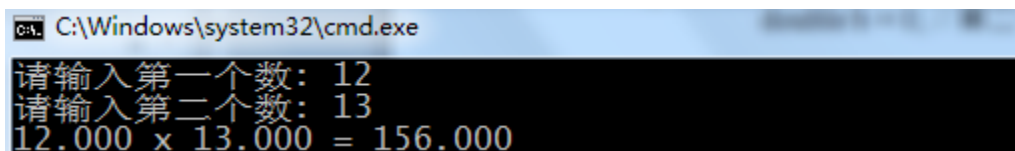
    printf("请输入第一个数: "); // 提示用户输入
    scanf("%lf", &a);
    printf("请输入第二个数: "); // 提示用户输入
    scanf("%lf", &b);

    double result = a * b;    //计算结果
    printf("%.3lf x %.3lf = %.3lf\n", a, b, result);
    return 0;
}
```

在这个程序中，首先定义了两个变量 `a,b` 用于存放小数。然后提示用户输入，当输入的数存到变量 `a,b` 里。然后用一个算式 `result = a * b` 来求出乘积，最后将算式和结果打印出来。

注：在 C/C++里，乘法符号是`*`，其具体原理和用法后续会讲，现在只要求会模仿使用即可。

下面是这个程序的输出显示：



```
C:\Windows\system32\cmd.exe
请输入第一个数: 12
请输入第二个数: 13
12.000 x 13.000 = 156.000
```

本书习题：本书配套在线题库（含答案及解析），请登录官网 <http://afanihao.cn> 查看。

本书相关资源，一律到官网获取：

- （1）视频教程：约 90 集，覆盖全部内容。建议与文字教程结合使用。
- （2）示例代码：本文字教程的全部示例源码。

3. 变量与常量

本章介绍常量和变量的概念。常量是不可以修改的量，变量是可以变化的量。介绍几种基本的变量类型：用于表示整数的类型 `bool/char/short/int`，用于表示小数的类型 `float/double`。

3.1 变量

变量表示可以变化的量。我们在下面的例子中，来观察变量的用法。（注意，前面的行号只是为了方便说明而加上去的，实际写代码的时候不需要输入行号）

////////// 例 CH03_A1 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int a = 1;
05      a = 2;
06      printf("a: %d \n", a);
07      a = 3;
08      printf("a: %d \n", a);
09      return 0;
10 }
```

在示例 CH03_A1 中，

第[04]行：定义一个变量 `a`，用于表示整数，初始值为 1；

第[05]行：将 `a` 赋值为 2，此时 `a` 的值变成 2；

第[06]行：将整数 `a` 打印出来；

第[07]行：将 `a` 赋值为 3，此时 `a` 的值又变成了 3；

第[08]行：再次将整数 `a` 打印出来；

现在，我们可以认识到变量就是一个可以变化的量，通过等号可以改变它的值。而对于代码中的其他行，读者暂时不必关心，随着学习的深入，会逐步给大家介绍。

下面我们具体讨论变量的相关语法细节。

3.1.1 变量的定义

定义一个变量时，要指定以下几个要素：①变量名 ②变量类型 ③ 初始值（可选）。

例如，下面一行代码就是定义了一个变量，

```
int abc = 10;
```

我们分析一下:

- ① 变量名: `abc`
- ② 变量类型: `int`, 表示它是一个整数类型
- ③ `10`: 指定其初始值为 `10`

另外, 记得末尾要加上一个分号, 否则就是语法错误。

下面我们再看一个例子, 在这个例子中定义了多个变量。

////////// 例 CH03_A2 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int a = 1;
05      int b;
06      double c = 10.1;
07      double d;
08      b = 2;
09      d = 10.2;
10      printf("a:%d b:%d c:%.3lf d:%.3lf\n", a, b, c, d);
11      return 0;
12 }
```

在示例 CH03_A2 中,

第[04]行: 定义了一个变量 `a`, 类型为 `int`, 表示整数, 初始值为 `1`;

第[05]行: 定义了一个变量 `b`, 类型为 `int`, 表示整数, 没有指定初始值;

第[06]行: 定义了一个变量 `c`, 类型为 `double`, 表示小数, 初始值为 `10.1`;

第[07]行: 定义了一个变量 `d`, 类型为 `double`, 表示小数, 没有指定初始值;

第[08]行: 将 `b` 赋值为 `2`

第[09]行: 将 `d` 赋值为 `10.2`

第[10]行: 将变量 `a, b, c, d` 的值打印显示

3.1.2 变量的命名

在 C/C++中, 变量的命名规则为: 必须是字母、数字、下划线的组合。可以用字母或下划线开头, 但不可以用数字开头。

下面是正确的变量名,

```
int name12 = 0;
int myage = 33;
int good_bye = 0;
```

下面是不正确的变量名,

```
int 12name = 0; // 错误! 不能以数字开头!
int my age = 33; // 错误! 不能含有空格! 只能是字母、数字、下划线的组合!
int good-bye = 0; // 错误! 不能含有横杠! 只能是字母、数字、下划线的组合!
```

3.1.3 变量的赋值

在 C/C++中, 使用等号=可以改变一个变量的值, 我们把这个操作称为“赋值”。

在下例中, 定义了变量 a 和 b, 然后使得赋值操作改变它们的值。

////////// 例 CH03_A3 //////////

```
#include <stdio.h>
int main()
{
    int a ;
    a = 1; // 将变量 a 赋值为 1
    a = -2; // 将变量 a 赋值为-2
    double b ;
    b = 1.2; // 将变量 b 赋值为 1.2
    b = -2.2; // 将变量 b 赋值为-2.2
    return 0;
}
```

3.2 整型变量

不仅 `int` 类型可以用于表示整数，还有一些其他的类型，也可以用于表示整数。

3.2.1 `char / short / int` 类型

最常用的三种整数类型为 `char`, `short`, 和 `int`, 它们都可以用于表示整数。例如,

```
char a = 12;
short b = 1280;
int c = 1280000;
```

它们的区别在于表示范围不同，一个 `char` 型变量只能表示从 -128 到 127 之间的整数，而一个 `int` 型变量则可以表示很大的数[-2147483648, 2147483647]。这意味着，当一个数比较小、位于 -128 到 127 之间时，我们可以用 `char` 型变量来表示；而超出了这个范围时，就不能用 `char` 型变量来表示。

在示例 CH03_B1 中，用一个 `char` 型变量表示 1224，再把变量的值打印显示出来。由于 1224 这个值超出来 `char` 所能表示的范围，所以打印显示的结果和我们预期的不一样。

```
////////// 例 CH03_B1 //////////
#include <stdio.h>
int main()
{
    char a = 1224;
    printf("value : %d \n", a);
    return 0;
}
```

对于我们初学者而言，我们大多数时间使用 `int` 型来表示整数，它是足够大的。在少数情况下，我们只需要记住 `char` 的表示范围就可以了。

当用 `scanf` 来输入一个整数时，只能用 `int` 类型，不能用 `char` 或 `short` 类型。例如，

```
////////// 例 CH03_B2 //////////
#include <stdio.h>
int main()
{
    char a = 0;
    scanf ("%d", &a); // 错！不能用 char 或 short 接收用户输入！只能用 int 型变量！
    return 0;
}
```

```
}
```

当用 `printf` 来输出一个整数时, `char`, `short`, `int` 型都可以输出。例如,

```
////////// 例 CH03_B3 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char a = 12;
```

```
    printf("a: %d\n", a); // 用 printf 可以输出 char 和 short 型变量
```

```
    return 0;
```

```
}
```

3.2.2 unsigned 无符号类型

无符号类型用于表示非负整数, 即大于或等于 0 的数。常用的类型为 `unsigned char`、`unsigned short` 和 `unsigned int`。

例如,

```
    unsigned char a = 12;
```

```
    unsigned int = 120900;
```

同样地, 三种无符号类型区别在于它们的表示范围。`unsigned char` 的表示范围最小, 是从 0 到 255。而 `unsigned int` 则最大, 从 0 到 4294967295。

对于初学者来说, 只需要记住 `unsigned char` 的表示范围即可。当我们要表示一个较大的正整数时, 直接用 `unsigned int` 即可。

无符号类型不能用于表示负数, 例如, 下面的代码是有问题的,

```
    unsigned char a = -12; // 不能表示负数
```

注: 事实上, 前面所学的 `char/short/int` 只是 `signed char/signed short/signed int` 的简写, 而关键字 `signed` 给省去了而已。至于 `unsigned` 与 `signed` 之间的本质联系, 参考本书附录《有符号整数与无符号整数》。

在用 `printf/scanf` 调用中, 无符号整数用 `%u` 作为控制符。同样地, 在用 `scanf` 接收输入时, 只能使用 `unsigned int`, 不能用 `unsigned char` 或 `unsigned short`。

```
////////// CH03_B4 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```



```
unsigned int a = 0;

scanf("%u", &a); // 输入时只能使用 unsigned int 来接收, 不能使用 unsigned char/short

printf("Got: %u \n", a);

return 0;

}
```

3.2.3 *bool 布尔类型

(*初学者请跳过本节。在第 20 章以后才会用到)

C++引入 `bool` 类型, 用于表示布尔型(boolean)数值, 它本质上是一种整数类型。只有两种取值: `true` 或 `false`, 注意 `true` 和 `false` 是字面常量, 它们是 C++关键字, 不是普通文本。

例如,

```
bool ready = true;

bool on = false;
```

`bool` 类型本质上等同于 `char`, 使用 `sizeof` 操作可以发现其大小为 1 字节。事实上, 字面常量 `true` 的值就是整数 1, `false` 的值就是整数 0。可以用 `printf` 来打印一下它们的值。

```
printf(" %d, %d \n", true, false);
```

程序员可以从不使用 `bool` 这个类型, 也能解决所有的问题。事实上, C/C++程序员一般用 `int` 或 `char` 来解决问题。例如,

```
int ready = 1; // 0 表示“假”, 非 0 表是“真”

int on = 0;
```

3.2.4 *enum 枚举类型

(*初学者请跳过本节, 具体内容见本书附录《枚举类型》)

3.2.5 *long long 长整数类型

(*初学者请跳过本节)

对于初学者来说, 32 位整型一般是够用的了 (`int` 和 `unsigned int`)。还有一种表示范围更大的整数类型, `long long` 和 `unsigned long long`, 它们是 64 位的整数、占 8 字节的内存。读者对此类型有点印象即可。

3.3 浮点型变量

用于表示小数的类型有两种: `double` 和 `float`, 通常称为浮点型(float-point)。

`double` 和 `float` 的主要区别也是表示范围不同。`float` 比 `double` 可以表示的范围要小得多, 但对于初学者来说, 只需要记住当需要表示高精度的小数时应该用 `double`; 当精度要求不高时 (7 位有效数字), 可以用 `float`。

下面的例子中, 定义了 `float` 型的变量,

```
float a = 3.14f; // 注意: 数字后面要加一个 f
```

```
float b = -87.9f; // 注意: 数字后面要加一个 f
```

在 `printf/scanf` 中, `float` 型 `"%f"` 作为控制符, `double` 型用 `"%lf"` 作为控制行。

例如,

```
float a = 10.135;
```

```
printf("%.2f ", a); // 输出 float 型
```

```
float b;
```

```
scanf ("%f", &b); // 从控制台接受输入
```

事实上, `lf` 代表的是 `long float-point`, 而 `f` 代表的是 `float-point`。

3.4 数的进制表示

在计算机领域, 常用二进制、十进制和十六进制来表示一个数。

3.4.1 数的十进制表示

日常生活中我们用的数都是十进制表示的。举例来说, 家里养了几只兔子, 要表示兔子的数量, 人们发明了几个符号: 0 1 2 3 4 5 6 7 8 9, 其中, 0 表示没有兔子, 9 表示养了九只兔子。

随着数量的增多, 当超过九只之后, 人们没有发明更多的符号来表示它, 而是用多位符号来表示一个数字。例如, 符号 "123" 表示的数量是一百二十三, 从数学上看, 每一个符号代表的权重是不一样的。举例来说:

$$123 = 1 * 100 + 2 * 10 + 3 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

$$9527 = 9 * 10^3 + 5 * 10^2 + 2 * 10^1 + 7 * 10^0$$

所谓十进制, 就是一共有十个符号序列, 逢十进一。每一位的权重是 10 的 $N-1$ 次方, N 表示其位置。这就是十进制的含义。

3.4.2 数的十六进制表示

十六进制和十进制本质上是同一个意思: 人们发明了十六个符号: 0 1 2 3 4 5 6 7 8 9 A B C D E F, 依次代表的兔子的数量为零、一、二、三、四、...、九、十、...、十五。逢十六进一, 所以再往上增长就是 10, 11, ..., 1A, 1B, ... 1F, 20, 21, ...

对于一个十六进制的表示的数，其每一位的权重是 16 的 N-1 次方。据此可以很容易地将一个十六进制的数换算成我们熟悉的十进制。

例如，

$$A2(x) = A * 16^1 + 2 * 16^0 = 162 \text{ (d)}$$

$$A2B3(x) = A * 16^3 + 2 * 16^2 + B * 16^1 + 3 * 16^0 = 41651 \text{ (d)}$$

为了区别不同的进制，我们在数字后面标注(x)表示十六进制（Hex, Hexadecimal），用 (d) 标识十进制（Decimal），用(b)标识二进制（Binary）。这只是我们在本书中自己定义的一种描述方式。

在 C/C++语法中，用 0x 或 0X 开头来表示十六进制的数。

例如，下面定义的两个整型变量的初始值都是 12。

```
int a = 12; // 默认为十进制
```

```
int b = 0x0C; // 当以 0x 开头时，以十六进制表示。
```

注意，可以写成 0x0C 或 0x0c，都是允许的。

3.4.3 数的二进制表示

在掌握了 16 进制之后，2 进制的问题也就迎刃而解，它们的数学原理都是一样的。对于 2 进制来说，人们发明了一共两个符号：0 和 1，再往上增长就要用多位符号来表示了，每一位的权重是 2 的 N-1 次方。因此，二进制数是这么增长的，例如从零到七的二进制表示：

0

10

11

100

101

110

111

根据每一位的权重，也能很容易地将二进制换算成我们所熟悉的十进制的数。

$$00000000 = 0 \text{ (十进制)}$$

$$00000001 = 1 \text{ (十进制)}$$

$$00001101 = 13 \text{ (十进制)}$$

$$11111111 = 255 \text{ (十进制)}$$

计算方法:

$$00001101 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

3.5 变量与内存

本节内容至关重要。

3.5.1 二进制存储

在计算机领域，数字最终是以二进制存储和表示的。这是因为二进制只有两个符号，容易用物理量来代表。

例如，一共设八个物理开关，其中√标识是开，否则为关。

√		√			√	√	
---	--	---	--	--	---	---	--

我们根据其状态对其进行数字化，把开当成 1，关当成 0，

1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---

这意味着，这组开关的物理状态可用于表示数字 10100110 (b)。

在计算机领域，把每一个开关称为位 (Bit)，八个位构成一个字节 (Byte)。显然，一个字节能够表示从 00000000(b)到 11111111(b)共 256 个数。

3.5.2 内存

内存用于存储数据的物理器件，拆开电脑，一般可以发现一块长方形的内存条。内存由很多个单元组成，每个单元可以认为是八个物理开关组成的，因此它可以表示一个八位的二进制数，即一个字节的数。

那么一般的内存条里有多少个单元呢？这个容易是很大的，一般以 GB 为单位。在计算机里，一般用 KB、MB、GB、TB (B 代表 Bytes) 这几个单位来衡量大小。它们的关系是：

$$1 \text{ KB} = 2^{10} \text{ B} , (\text{Kilo Bytes})$$

$$1 \text{ MB} = 2^{10} \text{ KB} , (\text{Mega Bytes})$$

$$1 \text{ GB} = 2^{10} \text{ MB} , (\text{Giga Bytes})$$

$$1 \text{ TB} = 2^{10} \text{ GB} , (\text{Tera Bytes})$$

其中 $2^{10}=1024$ ，近似为 1000，所以也近似等于日常生活中的 1，1000，1000000，1000000000 这样的千倍关系。

3.5.3 变量的大小

当程序运行时，每一个变量其实于都对应关一块内存。而变量的值，其实就是物理内存里那几个字节里存储的数据。

我们知道，一个 `char` 型变量可以表示的范围很小，只能表示 $[-128, 127]$ 。而一个 `int` 型变量的表示范围则很大，能表示一般对应了 4 个字节的内存，能表示 $[-2147483648, 2147483647]$ 。为什么有这么大差异呢？其本质原因是：在内存里，`char` 型变量占据了一个字节，而 `int` 型变量占据了四个字节。一个字节能表示的范围是 `00~FF`，只能表示 256 个数。四个字节表示的范围则能表示 `00000000~FFFFFFFF`，这是一个很大的范围。

我们把变量在内存里所占的字节数，称为变量的大小。在 C/C++ 语言里，可以用操作符 `sizeof` 来测量一个变量或类型的大小。在示例中，用 `sizeof` 来测量 `int` 类型的大小，其结果为 4。

////////// 例 CH03_C1 //////////

```
#include <stdio.h>

int main()
{
    int a = 0;

    printf("size: %d \n", sizeof(a)); // 用 sizeof 测量变量 a 的大小
    printf("size: %d \n", sizeof(int)); // 用 sizeof 测量类型 int 的大小

    return 0;
}
```

用同样的方法测量其他类型的大小，结果为：`char(1)`, `short(2)`, `int(4)`, `float(4)`, `double(8)`，显然地，如果一个类型占的字节越多，它能表示的范围就越大、精度就越高。例如，`double` 占了 8 个字节，而 `float` 只有 4 个字节，因而 `double` 要比 `float` 能存储更多的信息（存储更多的有效数字）。

3.5.4 变量在内存中的表示

我们以 `char` 型和 `int` 型变量为例，阐述变量在内存中的表示。

在代码中定义两个变量，为了方便描述，使用十六进制来表示数字。

```
char a = 0x1A; // 即十进制的 26
```

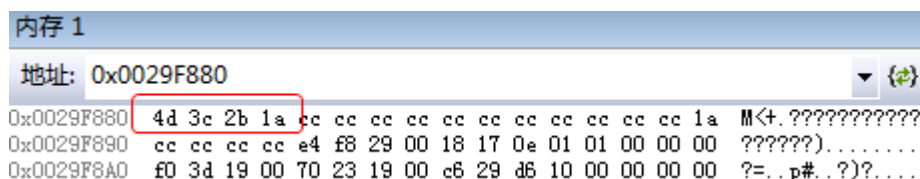
```
int b = 0x1A2B3C4D; // 即十进制的 439041101
```

变量 `a` 和 `b` 在内存中的表示为：



虽然我们不知道在一个容量巨大的内存条中，究竟是哪几个单元对应了变量 `a`, `b`。但我们可以肯定的是，至少是存在着一个单元对应于 `a`，存着四个单元对应于 `b`。

注：实际上，我们可以在 VC 中直接观测到变量在内存中的几个单元，具体方法参考本书附录《VC2008 调试方法》。例如，我们可以观察到变量 `b` 在内存中对应的那个字节，如下图所示，对应的几个字节已经被圈中标识：



3.5.5 变量的地址

我们对内存中的每个单元进行编号，从 0 开始依次增长，依次为 0, 1, 2, ..., 1000, 1001, ..., 1000000000, 1000000001, ... 把这个编号称为内存单元的地址。那么每个内存单元都有一个地址，用十六进制表示的话，地址范围是 00000000~FFFFFFFF。

在程序运行时，对于每一个变量，它在内存中都对应了几个内存单元。我们把它对应的内存单元的首地址（第一个字节的地址），称为变量的内存地址，简称为变量的地址。显然，变量的地址是一个整数。

通过操作符 `&` 可以取得变量的地址。在示例 CH03_C2 中，用 `&a` 取得 `a` 的地址，并用格式符 `%08X` 将这个地址以 16 进制形式打印显示。

```

//////////////////// CH03_C2 //////////////////////
#include <stdio.h>

int main()
{
    char a = 0x1A;
    int b = 0x1A2B3C4D;

    printf("a : address = %08X \n", &a); // &a 表示变量 a 的地址
    printf("b : address = %08X \n", &b); // &b 表示变量 b 的地址

    return 0;
}

```

3.5.6 理解变量的赋值

变量的值，其实就是这个变量对应的那几个内存单元的值。

因而，当我们在代码中对变量赋值的时候，其实修改了那几个内存单元的值。

```

//////////////////// 示例 CH03_C3 //////////////////////

```

```

01 #include <stdio.h>

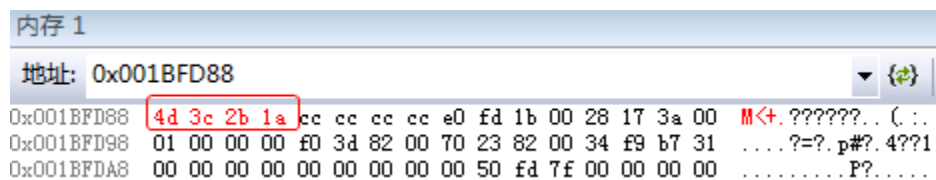
```

```

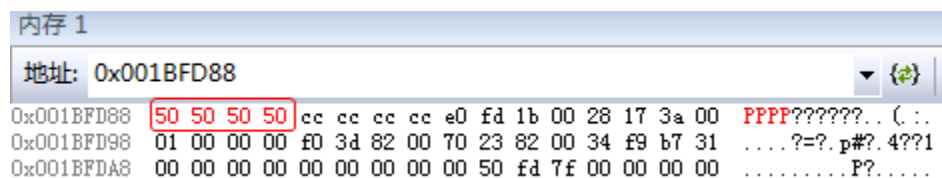
02  int main()
03  {
04      int b = 0x1A2B3C4D;
05      b = 0x50505050;
06      return 0;
07  }

```

程序执行第[04]行后, b 对应的内存的值为: (圈中所示)



程序执行第[04]行后, b 对应的内存的值发生了变化: (圈中所示)



3.6 const 常量

当在类型名前面加上关键字 `const` 后, 表示它是一个只读的量。这种变量不能修改它的值, 因而称为常量。例 CH03_D1 中, 试图对 `const` 常量 `a` 和 `pi` 进行赋值操作, 这是语法不允许的, 编译器会报错。

////////// 例 CH03_D1 //////////

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    const int a = 10;
```

```
    const double pi = 3.1415926;
```

```
    a = 11; // 错误! 不允许赋值操作!
```

```
    pi = 3.1415927; // 错误! 不允许赋值操作!
```

```
    return 0;
```

```
}
```

`const` 常量是只读的, 可以读取它的值, 或者用 `printf` 打印出来。

```
////////// 例 CH03_D2 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    const int MAX_SIZE = 1024;  
  
    printf("a: %d\n", MAX_SIZE); // 读取 const 常量的值  
  
    int b = MAX_SIZE; // 将 MAX_SIZE 的值赋值给变量 b  
  
    printf("b: %d\n", b);  
  
    return 0;  
}
```

3.7 *字面常量

(*初学者可以跳过本节)

字面常量 (Literal Constant)，指的是在代码里写在字面上的值。

例如，

```
int a = 123;
```

那么代码里的 123 字样，就是一个字面常量。

在例 CH03_D1 中，对我们已经认识的字面常量作了注释。

```
////////// 例 CH03_D1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    int a = 123; // 123 是字面常量  
  
    double b = 3.14159; // 3.14159 是字面常量  
  
    float c = 123.4f; // 123.4f 是字面常量  
  
    printf("%d %lf %f\n", a, b, c);  
  
    printf("I have %d numbers\n", 3); // 3 是字面常量  
  
    return 0;  
}
```


对于初学者来说,暂时不需要对字面常量有深度的理解。目前只需要记住一点:常量不可以被赋值。

例如,以下代码是有语法错误的:

```
#include <stdio.h>

int main()
{
    10 = 10; // 错误! 常量不能被赋值!
    12 = 13; // 错误! 常量不能被赋值!
    return 0;
}
```

3.7.1 字面常量的类型

C/C++是强类型的语言,所有的变量和常量都是有类型的。例如,12是int型,12.0是double型,12.0f是float型。

例如,在给变量赋值的时候,要注意赋值符左右两侧的类型是否匹配:

```
int a = 12;
double b = 12.0;
float c = 12.0f;
```

3.8 常用类型的范围

C/C++里常用的类型,及表示范围如下表所示:

类型	sizeof	表示范围	说明
char	1	-128~127	$-2^7 \sim (2^7 - 1)$
short	2	-32768~32767	$-2^{15} \sim (2^{15} - 1)$
int	4	-2147483648~2147483647	$-2^{31} \sim (2^{31} - 1)$
unsigned char	1	0~255	$0 \sim (2^8 - 1)$
unsigned short	2	0~65535	$0 \sim (2^{16} - 1)$
unsigned int	4	0~4294967295	$0 \sim (2^{32} - 1)$
float	4	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	7位有效数字
double	8	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$	15位有效数字
long long	8	很大	$-2^{63} \sim (2^{63} - 1)$

unsigned long long

8

很大

$0 \sim (2^{64} - 1)$

本书习题：本书配套在线题库（含答案及解析），请登录官网 <http://afanihao.cn> 查看。

本书相关资源，一律到官网获取：

- （1）视频教程：约 90 集，覆盖全部内容。建议与文字教程结合使用。
- （2）示例代码：本文字教程的全部示例源码。

4. 数组

本章介绍数组的语法和使用。数组用于表示若干个相同类型的量。

4.1 引例

下面例子将引出一个问题，本章的语法规则是用于解决这种问题。

设一个学生的成绩在 0~100 之间，则可以用 `char` 型变量来表示：

```
char a = 98;
```

设一个班有 30 个学生，如何表示他们的成绩呢？可以用 30 个 `char` 型变量来表示：

```
char a00 = 98;
```

```
char a01 = 95;
```

```
...
```

```
char a29 = 88;
```

一共定义 30 个变量，`a00~a29`，每个变量表示一位同学的成绩。这样虽然可以勉强完成数据的表示，但是有一个显然的问题：代码过于臃肿和机械。如果 30 个同学的成绩要用 30 个变量变量，那如果有 100 个同学，难道要定义 100 个不同的变量吗？

在 C/C++ 里，使用数组的语法，只需要定义一个变量，就可以表示 100 个同学的成绩：

```
int arr[100];
```

4.2 数组的定义

数组用于表示一组数值。例如，

```
char arr[5];
```

其中，`arr` 称为“数组变量”，简称“数组”。它表示 5 个 `char` 型数据。我们把每一个数据称为一个“元素”。

数据组的定义中包含以下几个要求：

- ① 元素类型：上例中元素类型为 `char`
- ② 元素的个数：中括号内指定个数，例如上面的数组长度为 5
- ③ 数组的名称：上例中数组的名称为 `arr`

数组的意义，是相当于把 N 个同类型的变量排列在一起。比如，对于 `char arr[5]` 就是说把 5 个 `char` 排列在一起。

同样的，我们定义基本类型的数组。例如，

////////// CH04_A1 //////////

```
char alpha[10]; // 10 个 char 型元素  
short years[20]; // 20 个 short 型元素  
int numbers[30]; // 30 个 int 型元素  
float scores[3]; // 3 个 float  
double values[12]; // 12 个 double 类型
```

4.2.1 数组的命名

数组的命名规则和变量名的规则相同，即“数字、字母、下划线的组合，但不能以数字开头”。

首先，变量的名字要符合其意义，这要求我们在命名时要能做到“词能达义，顾名思义”，不要给变量起一个不相关的名字。

其次，数组变量的名字一般使用小写字母。如果由多个单词组成，则中间以下划线分开。

注：你可以较为随意的定义一个变量名，但是这样“随意”地写代码，会使代码难以阅读，“可读性”降低。因此建议按照推荐的方式来给变量命名。

4.2.2 数组的长度必须是常量

数组的长度在中括号内指定，必须是一个整型常量。例如，

```
int arr[12];
```

不能用变量来表示一个数组的长度，例如，以下定义是错误的，

```
int size = 12;
```

```
int arr [ size ]; // 编译器报错！数组的长度必须是常量！
```

4.3 数组的基本方法

4.3.1 数组的初始值

可以定义数组的时候，指定每一个元素的初始值。例如，

```
char arr[5] = { 90,91,92,93,94 };
```

其语法要素为：

- ① 使用大括号，大括号末尾加上分号
- ② 大括号内指定初始值，每个初始值以逗号隔开，但最后一个数组末尾不加逗号

下面再介绍一些特殊的写法。

(1) 不指定初始值

定义时可以不初始化。例如，

```
char arr[5]; // 定义了一个长度为 5 的 char 型数组，不指定初始值
```

(2) 只指定一部分初始值

```
char arr[5] = {90,91 }; // 只指定前 2 个元素的初始值
```

注意：此种情况下，只能定义前几个元素的值。

如果在定义的时候只能给出后面几个元素的值，则必须手工地把前面的元素设置一个初始值，例如，

```
char arr[5] = {0,0,0, 90,91 }; // 只知道后 2 个元素的值，于是把前 3 个元素的值设为 0
```

(3) 只有初始值，没有长度

中括号的长度可以省略不写。当长度不写明时，编译器会根据初始化列表中的元素的个数来计算其长度。

```
char arr[ ] = { 90,91, 92, 93 }; // 中括号没有写明长度，编译计算得到其长度为 4
```

(4) 按位清零

```
char arr[5] = { 0 };
```

则所有元素的值都是 0

4.3.2 访问数组中的元素

例如，用一个数组变量 `values` 来存放 5 个 `int` 型整数，

```
int values[5] = {1, 2, 3, 4, 5};
```

则用 `values [0]`表示数组的第 1 个元素，

例如,

```
////////// CH04_B1 //////////  
  
    values[0] = 11; // 将第 1 个元素的值设为 11  
  
    printf("%d \n", values[0]); // 打印第一个元素的值
```

也就是说, 利用数组名加中括号可以访问数组中的每个元素, 中括号内的数值表示元素的位置(称为“下标”或“索引”)。元素的下标从 0 开始计算, 这意味着, 第一个元素使用 `values[0]`, 第二个元素使用 `values[1]`,, 以此类推。

例如, 下面的代码用于对 5 个元素求和,

```
// 对所有的元素求和  
  
int total = values[0] + values[1] + values[2] + values[3] + values[4];
```

例如, 下面的代码用于计算第一个数和第五个数的差值

```
// 第一个数和第五个数的差值  
  
int delta = values[0] - values[4];
```

4.3.3 用 `sizeof` 取得数组的大小

数组的大小由元素的类型和元素的个数共同决定。例如,

```
int arr[100];  
  
int size = sizeof(arr); // 大小为 100*4
```

4.4 数组的内存视图

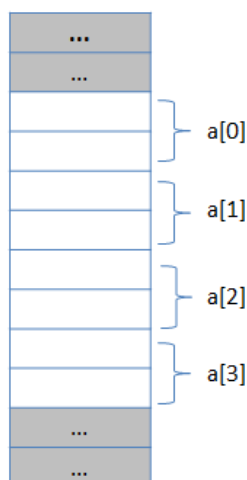
在第 3 章中我们知道, 每一个变量都与一块内存区域对应, 修改变量的值实质上就是修改对应内存的值, 而读取变量的值就是读取相应内存的值。

对于数组来说, 它也是变量, 而且相当于若干个基本变量排列在一起。那么很容易理解, 数组在内存视图角度来看, 是直接对应了若干个内存单元。

定义一个 `short` 数组

```
short a[4] = { 0x1111, 0x2222, 0x3333, 0x4444 };
```

由于一个 `short` 占 2 个字节, 所以 4 个 `short` 就占了 8 个字节的内存。在内存中, `a[0], a[1], a[2], a[3]` 可以视为连续排列的 8 个 `short` 型变量。下面的示意图阐述了这个概念。



注：在 VC 的调试状态下的“内存”窗口中，可以直接观看数组 a 对应的内存。如下图所示，方框线内的 8 个字节就是数组 a 所占据的内存。其中 11 11 是 a[0]，22 22 是 a[1]，33 33 是 a[2]，44 44 是 a[3]。可以很直观的看到，它们是紧密排列的。

内存 1		
地址:	0x0017FAEC	
0x0017FAEC	11 11 22 22 33 33 44 44	dc cc cc cc f2 54 05 11 ..""33DD?????T..
0x0017FAFC	4c fb 17 00 48 17 e5 00 01 00 00 00 f0 3d 3f 00	L?...H?...?=?..
0x0017FB0C	70 23 3f 00 42 55 05 11 00 00 00 00 00 00 00 00	p#?.BU.....

4.5 常见问题

(1) 初始化列表的长度不能大于数组的长度

在下面的代码中，数组中的长度为 5，大括号却有 6 个初始值。编译器报错。

```
int values [5] = { 1,2,3,4,5,6 }; // 错误: 初始化列表的长度不能大于数组长度
```

(2) 越界访问

对于一个长度为 N 的数组，其下标的合法范围是从 0 到 N-1。一个不合法的下标访问称为“越界”访问。

例如，

```
int values [ 5] = { 1, 2,3 ,4, 5};
```

```
values[5] = 123; // 越界
```

对于上述代码，编译器无法检查错误，但在运行时会出错，通常程序崩溃，或者出现其他意想不到的错误。

对于由编译器能够检查出来的错误，称为“编译错误”，或“语法错误”。

当编译成功、但运行时出错的错误，称为“运行时错误”。

对于初学者来说，遇到的经常是“编译错误”，这是因为大家还不熟悉基本语法。但是当大家熟悉基本语法之后，大家将来经常要面对的都是“运行时错误”。

4.6 数组的使用实例

4.6.1 实例 1

定义一个数组，存放 5 个 int 型数据，并将初始值设为 1，2，3，4，5。

////////// 例 CH04_C1 //////////

```
int b[5];
b[0] = 1;
b[1] = 2;
b[2] = 3;
b[3] = 4;
b[4] = 5; // 正确! 按索引赋值
```

4.6.2 实例 2

定义另一个数组，其中元素的值与上一个例子中的值相反。

////////// 例 CH04_C2 //////////

```
int a[5] = {1,2,3,4,5};
int b[5];
b[0] = a[4];
b[1] = a[3];
b[2] = a[2];
b[3] = a[1];
b[4] = a[0];
```

4.6.3 实例 3

交换第二个元素和第三个元素的值。

////////// 例 CH04_C3 //////////


```
int a[5] = {1,2,3,4,5};  
int t = a[1]; // 用 t 记录第二个元素的值  
a[1] = a[2]; // 改变第二个元素的值  
a[2] = t;    // 改变第三个元素的值
```

4.7 多维数组

二维数组和更高维的数组在实践工程中很少用到，不是本章的重点。

4.7.1 二维问题的表示

有些问题用一维数组难以表示。例如，现有在一个 4x4 的棋盘，每个单元格里放一个数字。使用一维数组表示：

```
int array[16];
```

用(x,y)表示其坐标的化，(3,4)表示第 3 行第 4 行。那么在(3,4)的位置对应的元素是 array[11]，对(3,4)操作也就是对 array[11]进行操作。其坐标与索引的转换公式是：

$$\text{index} = (x-1) * 4 + (y-1)$$

显然，这样表示不直观，阅读起来有些吃力，访问一个元素还先用公式转换先把坐标转换成索引才行。

在 C++中，可以用 2 维数组来表示这个问题。下面的代码中，直接用一个二维数组来定义棋盘，

```
int array[4][4];  
array[2][3] = 128; // 对坐标(3,4)位置进行赋值
```

4.7.2 二维数组的定义

二维数组的定义：

```
type name [ N1 ] [ N2 ] ;
```

其中，type 是元素类型, name 是数组变量的名称，N1 是第一维的大小，N2 是第二维的大小。

可以用行、列的概念来理解二维数组。第一个下标是行号，第二下标是列号。

例如，对于一个 4x3 的表格 a[4][3]，其下标依次是

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]

至于 3 维数组、4 维数组以至于 N 维数组，完全可以类似得到。由于高维数组并不常用，这里不再赘述。

4.7.3 二维数组的初始化

二维数组仍然可以大括号初始化，由于有两维，所以可以用二层大括号来分别初始化每行。下面的代码中，对一个 4 行 3 列的数组进行初始化，

```
int a[4][3] =  
{  
    { 11, 12, 13 },  
    { 21, 22, 23 },  
    { 31, 32, 33 },  
    { 41, 42, 43 }  
};
```

4.7.4 二维数组的本质

二维数组，乃至三维、四维等高维数组，只是在形式上比一维数组更直观更容易操作，其本质仍然是一维数组。可以从内存视频清楚地看出这个结论。

比如，对于 `int a[4][3]`，在内存中对应连续的 12 个 `int`：`a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]` ... `a[3][2]`（注意和一维数组的对应顺序）。

注：千万不要误以为二维数组在内存中它是矩阵方式排列的。实际上，它在内存中也是紧凑排列的若干个元素，只是编译器帮了个忙，它帮忙把二维坐标转换成一维坐标了。

本书习题：本书配套在线题库（含答案及解析），请登录官网 <http://afanihao.cn> 查看。

本书相关资源，一律到官网获取：

- （1）视频教程：约 90 集，覆盖全部内容。建议与文字教程结合使用。
- （2）示例代码：本文字教程的全部示例源码。

5. 字符与字符数组

本章介绍如何表示一个字符，以及如何表示一串字符。

5.1 字符是什么

观察键盘上的按键，要以把字符分为几类：

字母：a b c ... z

数字：0 1 2 ... 9

标点：+ - * / ; , . 等等

控制字符：Tab, Enter 等等

把这几十字符分别用一个数字与之对应，表示这种对应关系的表称为 ASCII 码表。把一个字符对应的那个数字，称为该字符的 ASCII 码。例如，在下表中列出小写字母('a' ~ 'z')，大写字母('A' ~ 'Z')，数字 ('0' ~ '9') 的 ASCII 码。可以查看本书附录《ASCII 码表》来查看全部规定。

'0' ⇔ 48
'1' ⇔ 49
...
'9' ⇔ 57
...
'A' ⇔ 65
'B' ⇔ 66
...
'Z' ⇔ 90
...
'a' ⇔ 97
'b' ⇔ 98
...
'z' ⇔ 122
...

注：ASCII 的全称是 American Standard Code for Information Interchange，美国标准信息交换代码，该表由国际标准组织制定。

5.2 字符的表示

在计算机里，所有数据必须以数字的形式表示，字符也不例外。

根据 ASCII 码表的规定，每个字符一个数字表示，而这个数字在 0-127 之间。在 C/C++里，char/short/int 都可以表示整数，由于字符的数值范围较小，我们选用 char 型变量来代表字符。

例如，

```
char ch = 65; // 则 ch 代表的是大写字母'A'
```

```
char ch2 = 49; // 则 ch 代表的是数字'1'
```

显然, 如果用 `short` 或 `int` 来表示字符的话就显得大材小用, 浪费空间。

5.3 字符的显示

可以使用 `printf` 将一个字符显示到控制台, 使用的格式化字符串为 `%c`。

例如, 下面的代码用于将 65 所代表的字符显示出来 (即大写字母 A) :

```
printf("Got: %c \n", 65 );
```

运行的显示如下图所示:



例如, 下面的代码将显示 49 所代表的字符 (即数字 1) :

```
char ch2 = 49;
```

```
printf("Show: %c \n", ch2 );
```

例如, 我们打印显示 "1+2" 这个字符串:

```
printf("%c%c%c \n", '1', '+', '2' );
```

例如, 我们打印几个空格字符:

```
printf("%c%c%c \n", 'A', ' ', ' ');
```

其中, 空格字符 ' ' 就是在单引号中间加上一个空格。

5.4 字符常量

在 C/C++ 代码中, 直接用字符常量来代表一个字符的 ASCII 码。使用单引号表示。例如,

```
char ch = 'A';
```

则 'A' 就是字符常量, 它是一种字面常量, 表示一个整数 65。

字符常量在任何时候都和它的 ASCII 码是等价的。也就是说, 虽然在形式上它是写成了 'A', 但编译器在处理代码的时候总是把它当成 65 来处理的。

所以，以下几种写法是等价的：

```
printf("Got: %c\n", 65 );  
printf("Got: %c\n", 'A');  
printf("Got: %c\n", 0x41);
```

由于字符常量完全等价于一个整数，所以我们可以这么写：

```
char ch1 = 'A' + 1; // 结果为 66  
char ch2 = 'B' - 1; // 结果为 65  
char ch3 = 'C' - 'A'; // 结果为 2
```

显然我们也可以用 `int` 和 `short` 来表示字符：

```
int ch1 = 'A';  
short ch2 = '9';
```

下面的代码用于显示字符 'Y' 的 ASCII 码：

```
printf (" %d\n", 'Y'); // 'Y' 是一个整数，所以可以用 %d 显示出来
```

5.5 字符数组

在 C/C++ 里，用一个 `char` 型数组来表示一串字符，称为“字符数组”。把这一串字符称为“字符串”。

字符两种初始化方法：

(1) 像普通数组一样初始化

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

(2) 特殊的初始化方法

```
char str[6] = "hello";
```

使用第 (2) 种方式在代码上要简洁一些。这两种方法的最终结果一样，就是在占据了 6 个字节的内存，其值依次值：

68	65	6C	6C	6F	00
----	----	----	----	----	----

在本教程中，有时也会表示成下面的样子，它们是同一数据不同表示方法，本质上一样。

h	e	l	l	o	\0'
---	---	---	---	---	-----

需要注意的时，当用字符数组来存储字符串时，必须以'\0'结尾。我们把'\0'称为字符串的结束符，它的 ASCII 码数值为 0。

5.5.1 输出字符串

可以用 `printf` 来向控制台输出一个以 0 结尾的字符串，使用的格式符为 `%s`。例如，

```
char str[6] = "hello";  
printf("string: %s \n", str);
```

5.5.2 输入字符串

我们可以使用 `gets` 来获取一个字符串。在示例 `CH05_C1` 中，使用 `gets` 将用户输入的字符串存入到数组 `buf` 中。注意，数组的长度要足够大，以免用户输入太长的字符串。

////////// 例 CH05_C1 //////////

```
#include <stdio.h>  
  
int main()  
{  
    /* 用户在控制台输入一个字符，  
       按回车结束，  
       所有输入的字符被存储到 buf 数组中  
    */  
  
    char buf[128]; // 定义一个足够大的数组  
  
    gets(buf);  
  
    // 把刚才输入的内容再打印输出  
  
    printf("Got: %s \n", buf);  
  
    return 0;  
}
```

注：对于 `gets` 的使用，我们现在只需要学会其简单的用法即可，暂是不必知道其语法原理。
在使用上：①用于接入收入的数组 `buf` 要足够大 ②`gets(buf)` 的小括号里是数组名字。

5.5.3 理解结束符的作用

字符串的末尾必须为一个数字 0 作为结束符，它是一个字符串结束的标识。如果一串字母不以 0 结束，那它就不算是一个有效的字符串。

例如，下面定义了一个字符数组，前 3 个字符是 b,a,d，但末尾并不是 0，那它不是一个有效的字符串，在用 printf 输出的时候会有问题：

```
char buf[4] = { 'b', 'a', 'd', -1 };
printf("Got: %s\n", buf);
```

控制台输出时，会打印出一些乱码字符，比如“烫烫烫”这样的乱码，这表明我们要打印的字符串没有以 0 结尾，是一个无效的字符串。如下图所示：



'\0'字符决定了字符串的长度，'\0'后面的字符不会被 printf 打印出来。正是从这个意义上，才把'\0'称为结束符的。例如，下面定义一个 9 字节的 char 数组，

```
char str[9] = { 'G', 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 };
printf("str: %s\n", str);
```

其输出结果为 Good，而后面的几个字符 Bye 是不会被显示输出的，属于无效内容。

我们把结束符'\0'前面的有效字符的个数，称为字符串的长度。也就是说，从第一个字符开始往后数，一直到达结束符 0，中间的非 0 字符的个数，就是该字符串的长度。

例如，计算下面几个字符串的长度。

```
char str1[9] = { 'G', 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 }; // 长度为 4
char str2[9] = { 0, 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 }; // 长度为 0
```

5.5.4 字符串的截断

利用字符串的结束符，可以轻易的截断一个字符串，例如，

```
char buf[32] = "hello,world";
buf[5] = 0;
printf("Result: %s\n", buf);
```

由于 buf[5]被设置成了结束符，所以字符串 buf 的有效部分就是"hello"了。

5.5.5 常见问题

(1) 字符数组不够大

例如，下面的字符数组不够存储 5 个字符+1 个结束的，至少要 6 个字节才够存储。

```
char buf[5] = "hello"; // 数组不够大，至少为字符串长度+1 才够
```

5.6 转义字符

5.6.1 转义字符的概念

如果我们想换显示，怎么做到呢？可以使用一些特殊的字符，如'\n'，表示换行。在下面的代码中，输出一个字符'1'，换行，再输出一个字符'2'，再换行。

```
printf("%c%c%c%c", '1', '\n', '2', '\n');
```

需要注意的是，以反斜杠开头的'\n'表示单个字符，其 ASCII 码为 10（0x0A）。

在 C/C++语言里，存在着用反斜杠开头的一些特殊的字符常量，它们承担特殊的控制功能，称为“转义字符”。

常见的转义字符有：

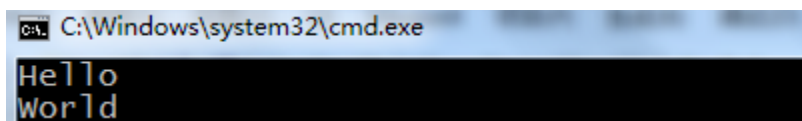
转义字符	值	意义
\a	0x07	(alert)响铃警告
\n	0x0A	(nextline)换行
\t	0x09	(tab) 制表位
\b	0x08	(backspace)回退
\r	0x0D	(return) 回车
\\	0x5C	反斜杠\
\"	0x22	双引号"
\0	0x00	字符串结束符

5.6.2 转义字符的使用举例

(1) 换行符

```
printf("Hello \nWorld \n");
```

输出显示如下，

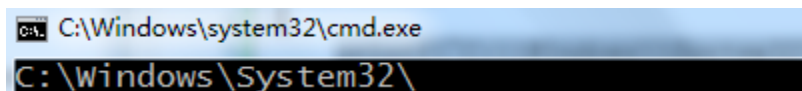


```
C:\Windows\system32\cmd.exe
Hello
World
```

(2) 反斜杠

```
printf("C:\\Windows\\System32\\ \n");
```

这在表示 windows 的文件路径时经常用到, 输出显示如下 ,

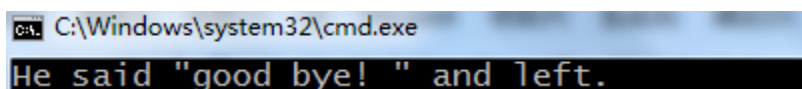


```
C:\Windows\system32\cmd.exe
C:\Windows\System32\
```

(3) 双引号

```
printf("He said \"good bye! \" and left.\n");
```

输出显示如下,



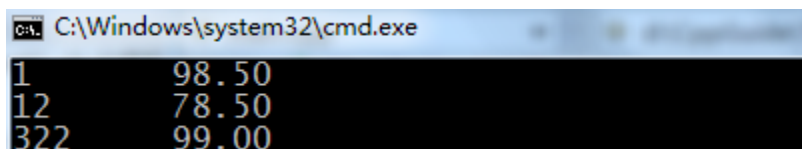
```
C:\Windows\system32\cmd.exe
He said "good bye! " and left.
```

(4) 制表符

下面的代码, 使用\t制表符, 使得输出的数据纵向按制表位对齐,

```
printf("%d\t%.2f\n", 1, 98.50f);
printf("%d\t%.2f\n", 12, 78.50f);
printf("%d\t%.2f\n", 322, 99.00f);
```

输出显示如下,



```
C:\Windows\system32\cmd.exe
1          98.50
12         78.50
322        99.00
```

本书习题: 本书配套在线题库(含答案及解析), 请登录官网 <http://afanihao.cn> 查看。

本书相关资源, 一律到官网获取:

(1) 视频教程: 约 90 集, 覆盖全部内容。建议与文字教程结合使用。

(2) 示例代码: 本文字教程的全部示例源码。

6. 表达式与操作符

表达式(expression)是一个式子, 它有一个值。它是操作数(operand)与操作符(operator)的组合。本章介绍如何使用各种类型的操作符来构造表达式。

6.1 算术表达式

由算术运算符连接起来的表达式, 称为算术表达式。算术操作符有 5 个, 如下表所示。

操作符	说明
+	相加
-	相减
*	相乘
/	相除
%	取模, 只适用于整数

其中, 相加、相减、相乘都和普通的算术概念一致, 对于整型和浮点型都适用。

例如:

```
3 + 4; // 其值为 7
```

```
3 - 4; // 其值为-1
```

```
3 * 4; // 其值为 12
```

```
0.2 + 0.4; // 其值为 0.6
```

```
0.2 - 0.4; // 其值为-0.2
```

```
0.2 * 0.4; // 其值为 0.08
```

对于整数来说, 相除是取倍数, 取模是取余数。

例如,

```
22 / 5; // 其值为 4 (22 是 5 的 4.4 倍, 结果只取整数部分)
```

```
22 % 5; // 其值为 2 (22 除以 5, 余数为 2)
```

对于浮点数来说, 相除后仍然是浮点数(保留小数部分)。浮点数不能进行取模运算。

```
22.0 / 5.0; // 其值为 4.4
```

123.5 % 10 ;//编译器报错! 浮点型不能进行模运算!

组合在一起的时候, 其优先顺序是 $*/\%+-$ 的顺序。当然, 如果记不清的话, 直接用括号 $()$ 来显式地指定结合顺序。

$a + b * 10 / c \% d - e$

6.2 赋值表达式

用等号可以对变量进行赋值。等号在这里被称为赋值操作符。例如:

$a = 10;$ // 其值是 10

$b = 11.23;$ // 其值是 11.23

需要特别注意的是, 在赋值表达式中, 整个表达式本身的值就是变量最终的值。用以下代码检查一下:

```
int a;
printf("test: %d \n", a=10); // 打印输出为 10,
```

6.2.1 左值

在 C/C++ 中, 可以放在等号左值的值, 称为左值。一般来说, 一个左值最终肯定对应一块内存, 修改它的值其实就是修改了那一块内存的值。

(1) 变量可以被赋值, 它是左值。由于变量对应一块内存, 修改变量的值, 就是修改了内存的值。

(2) 数组元素可以被赋值, 它也是左值。数组元素与对应着内存, 例如 $\text{arr}[0] = 123$ 表示修改第一个元素对应的内存的值。

(3) 字面常量不能被赋值, 它不是左值。例如, 下面的代码是错误的。

$1000 = a;$ // 编译器报错

6.2.2 理解赋值运算

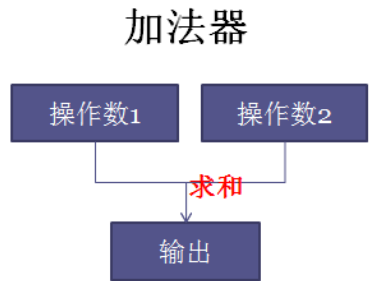
赋值运算在具体操作时有两步:

- ① 计算出等号右侧的表达式的值
- ② 将值写入到左侧变量对应的内存中

下面, 举一个例子, 来解释这一过程。

```
int a = 1;
int b = 2;
a = a + b; // 此行为赋值运算
```

我们看一下这一步赋值运行的具体过程。首先，我们知道 CPU 中存在一个加法器单元，其功能是实现加法运算。只要将两个数传给它，它就能执行运算，求出两个数的和，并把结果放在输出单元里。



在 CPU 执行 $a = a + b$ 时，

① 求右侧表达式 $a + b$ 的值

从 a 取值送到“操作数 1”单元，从 b 取值送到“操作数 2”单元。得到结果，存在“输出”单元。

② 从加法器的“输出”单元取值，存放 a 对应的内存。

6.2.3 赋值与算术运算合并

算术运算可以和赋值运行简写，于是增加了一些简写的操作符，如下表所示。

操作符	说明
$+=$	$a += b$ 相当于 $a = a + b$
$-=$	$a -= b$ 相当于 $a = a - b$;
$*=$	$a *= b$ 相当于 $a = a * b$;
$/=$	$a /= b$ 相当于 $a = a / b$;
$\%=$	$a \% = b$ 相当于 $a = a \% b$;

6.2.4 等号串连的写法

多个赋值可以串连在一行内。例如，

$a = b = c = 10$;

相当于

$(a = (b = (c = 10)))$;

6.3 关系表达式

关系表达式是用于表示两个数的大小关系的式子。关系表达式有关系操作符连接起来。

像

`10 > 9`

`a < b`

等就是关系表达式，而大于号`>`和小于号`<`就是关系操作符。

操作符	说明
<code><</code>	小于
<code><=</code>	小于或等于
<code>></code>	大于
<code>>=</code>	大于或等于
<code>==</code>	等于
<code>!=</code>	不等于

关系表达式的值有两种: 1（真）或 0（假），也就是说要么为真、要么为假。在 C++中以零表示假，以非零表示为真。例如：

```
10 > 9; // 真，其值为 1
```

```
10 > 11; // 假，其值为 0
```

可以用 `printf()`函数来检查一下这个结论：

```
printf("test: %d \n", 10 > 11); // 打印显示为 0
```

```
printf("test: %d \n", 11 > 10); // 打印显示为 1
```

由于表达式是右值，因此可以赋值给变量。

```
int value = 9 > 10;
```

```
printf(" values is : %d \n", value); // 打印结果: 0
```

6.4 条件表达式

条件表达式，也称为问号表达式，它是由条件操作符构造成的、带判断条件的表达式。其一般形式为，

`expr1 ? expr2 : expr3`

意义：

当 `expr1` 为“真”时，表达式的值为 `expr2`

当 `expr1` 为“假”时，表达式的值为 `expr3`

例如，

`0 ? 11 : 22 ; // 条件为假（0），整个表达式的值为 22`

`1 ? 11 : 22 ; // 条件为真（非 0），整个表达式的值为 11`

`1 ? 11 : 22 ; // 条件为真（非 0），整个表达式的值为 11`

`(a = 10) ? 11 : 22 ; // 条件值为 10（非 0），整个表达式的值为 11`

`(3 + 4) ? 11 : 22 ; // 条件值为 7，整个表达式的值为 11`

可见，在阅读条件表达式的时候，第一步就是计算一下问号前面的条件的值。如果条件值为真（非 0），则取冒号前面的值；如果条件值为假（0），则取冒号后面的值。

下面列出几个例子：

`printf ("%d \n", 0 ? 100 : 200); // 假，显示 200`

`printf ("%d \n", 1 ? 100 : 200); // 真，显示 100`

`printf ("%d \n", -2 ? 100 : 200); //真，显示 100`

`printf ("%d \n", (5 + 1) ? 100 : 200); //真，显示 100`

又如，用 `int` 型变量 `score` 来表示一个学生的成绩，当大于等于 60 时，显示 T；否则显示 F。代码实现如下：

```
int score = 90;
printf ("%c \n", score > 60 ? 'T': 'F');
```

再例如，当成绩在 90 以上为 A，60 以下时为 C，中间的为 B。代码实现如下：

```
int s = 80;
char result = (s > 90 ? 'A': ( s<60 ? 'C': 'B' ));
```

```
printf ("%c \n", result);
```

6.5 逻辑表达式

操作符	说明
!	逻辑非
&&	逻辑与
	逻辑或

6.5.1 逻辑非

其语法形式为:

```
! expr
```

即, 在表达式之前加一个感叹号, 表示对该表达式取非。具体操作是, 如果 `expr` 的值是“真”, 则取非后变成“假”(0); 如果 `expr` 为“假”, 则取非后为“真”(1)。

```
printf("%d \n", !23); // 打印结果: 0
printf("%d \n", !0);  // 打印结果: 1
printf("%d\n", !( 10 > 9 )); // 打印结果: 0
```

6.5.2 逻辑与

逻辑与, 表示“并且”的意思。其形式为:

```
expr1 && expr2
```

判断规则: 当 `expr1` 为真、并且 `expr2` 为真时, 结果为真; 否则为假。

例如:

```
2 && -1; // 真真=> 真
1 && 0;  // 真假=> 假
0 && 4;  // 假真=> 假
0 && 0;  // 假0=> 假
3>4 && a; // 假*=> 假 (不论 a 是真是假, 结果总是假)
```

提前结算: 当 `expr1` 为假时, 结果已经明了, 此时 `expr2` 不会被计算。

例如,

```
int a = 4;
```

```
int b = (3 > 4) && (a=123); // 提前结算, 右式 a=123 不会被执行, 所以 a 值不变
```

6.5.3 逻辑或

逻辑或, 表示“或者”的意思。其形式为:

```
expr1 || expr2
```

判断规则: 当 `expr1` 为真、或者 `expr2` 为真时, 结果为真; 否则为假。

例如:

```
3 || 2; // 真真=>真
```

```
1 || 0; // 真假=>真
```

```
3 < 4 || 2; // 假真=>真
```

```
0 || 0; // 假假=>假
```

```
4 > 3 || a; // 真* => 真 (不论 a 是真是假, 结果总是真)
```

提前结算: 当 `expr1` 为真时, 结果已经明了, 此时 `expr2` 不会被计算。

例如,

```
int a = 4;
```

```
int b = (5 == 5) || (a=123); // 提前结算, 右式 a=123 不会被执行, 所以 a 值不变
```

6.6 逗号表达式

逗号表达式是从逗号分开的一组表达式, 计算的时候从左到右依次计算, 最后一个表达式的值为整个表达式的值。

```
ex1, ex2, ex3, ..., exN
```

其中, `exN` 是一个表达式。

逗号表达式, 是以逗开隔开的一系列表达式。其形式为:

```
expr1, expr2, expr3, ..., exprN
```

其运算规则是, 从左到右依次计算每个子式的值, 并把最后一个子式的值作为整个表达式的值。

例如,

```
int a = 4;

printf("%d\n", (a+12, !3, a=1)); // 输出为 1, 因为最后一个子式 a=1 的值为 1
```

又如,

```
int nnn = (1, 2, 3, 4);

printf("nnn = %d\n", nnn); // 打印结果: 4
```

又如,

```
int a = 2;

int b = 3;

int nnn = (a+b, a-b, a*b);

printf("nnn = %d\n", nnn); // 打印结果: 6
```

6.7 自增/自减操作符

++是自增操作符, 表示对变量加 1; --是自减操作符, 表示对变量的值减 1。只能作用于整型变量, 不能作用于浮点型变量。

例如:

```
int a = 10;

a++; // 相当于 a=a+1

a--; // 相当于 a=a-1

++a; // 相当于 a=a+1

--a; // 相当于 a=a-1
```

6.7.1 前置

前置时, 先执行自增/自减操作, 再执行所在行的表达式。

```
int a = 10;

printf("%d", ++a);
```

相当于:

```
int a = 10;

a += 1;

printf("%d", a);
```

6.7.2 后置

后置时, 先计算本行的表达式, 之后再执行自增操作。

```
int a = 10;

printf("%d", a++);
```

相当于:

```
int a = 10;

printf("%d", a);

a += 1;
```

注: 自增和自减操作符只要简单掌握即可, 它们不是 C/C++语法的重点, 甚至完全不用它们没有关系。在 C/C++中, 要尽量把代码写得简单易读, 不要在一行内大量使用自增/自减操作符。

6.8 *位操作符

(*初学者可以跳过本节, 这不会影响到后续的学习)

操作符	说明
~	按位取反
<<	左移
>>	右移
<<=	左移并赋值
>>=	右移并赋值
&	按位与
^	按位异或
	按位或
&=	按位与赋值

$\wedge=$	按位异或赋值
$ =$	按位或赋值

所有的位操作只适用于整数，即 `char`, `short`, `int`, `unsigned char`, `unsigned short`, `unsigned int`。更具一点，只有无符号整数才适合使用位操作。

6.8.1 按位表示

首先，一个字节由 8 个位组成，这里先以最短的整型 `unsigned char` 来说明位的含义和用法。例如，下面定义了 2 两个变量 `M` 和 `N`，

```
unsigned char M = 0xA7, N = 0xE3;
```

其中，`M` 的按位表示为 (最左侧为高位 bit7，最右侧为低为 bit0)

1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

`N` 的按位表示为(最左侧为高位 bit7，最右侧为低为 bit0)

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

可以类推一下，`unsigned short` 就是 16 个位，`unsigned int` 是 32 位。

6.8.2 位运算规则

需要理解与、或、取反、异或这 4 种位操作。两个位 `A` 与 `B` 进行位运算是，它们的运行规则是：

“与”运算： `A=1` 且 `B=1` 时，所得结果是 1；否则为 0

1	&	1	=	1
1	&	0	=	0
0	&	1	=	0
0	&	0	=	0

“或”运算： `A=1` 或 `B=1` 时，所得结果是 1；否则为 0

1		1	=	1
1		0	=	1
0		1	=	1
0		0	=	0

“异或”运算： `A` 与 `B` 不同时，所得结果是 1；否则为 0

1	^	1	=	0
---	---	---	---	---

$$\begin{array}{rcl} 1 & \wedge & 0 = 1 \\ 0 & \wedge & 1 = 1 \\ 0 & \wedge & 0 = 0 \end{array}$$

“取反”运算：1 取反得 0；0 取反得 1

$$\begin{array}{l} \sim 1 = 0; \\ \sim 0 = 1; \end{array}$$

现在利用上面的规则，对 M 与 N 进行位运算，

与运算：M & N = ?

$$\begin{array}{rcl} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 = 0xA7 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 = 0xE3 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 = 0xA3 \end{array}$$

或运算：M | N = ?

$$\begin{array}{rcl} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 = 0xA7 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 = 0xE3 \\ \hline 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 = 0xE7 \end{array}$$

异或运算：M ^ N = ?

$$\begin{array}{rcl} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 = 0xA7 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 = 0xE3 \\ \hline 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 = 0x44 \end{array}$$

取反运算：~M = ?

$$\begin{array}{rcl} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 = 0xA7 \\ \hline 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 = 0x58 \end{array}$$

可以自己编写代码，测试一下运算的结果，与上面计算的结果比较。

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    unsigned char M = 0xA7, N = 0xE3;
```

```
    unsigned char result;
```

```
    result = M & N;
    printf("%02X \n", result);
    result = M | N;
    printf("%02X \n", result);
    result = M ^ N;
    printf("%02X \n", result);
    result = ~M;
    printf("%02X \n", result);
    return 0;
}
```

在掌握了&|^~四种位操作符之后，&=，|=，^=就比较容易理解和掌握了。

M &= N; 相当于 M = M & N

M |= N; 相当于 M = M | N

M ^= N; 相当于 M = M ^ N

6.8.3 移位操作

下面再看一下移位运算。再看一下 M(0xA7)的表示：

1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

(1) 右移：

M >> 1 表示 M 的所有位右移一位，左侧填充 0

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

M >> 3 表示 M 的所有位右移三位，左侧填充 0

0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---

M >> 8 表示 M 的所有位右移八位，左侧填充 0（原有 8 个位全部移走，剩下的填充为 0）

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

(2) 左移:

$M \ll 1$ 表示 M 的所有位左移一位, 右侧填充 0



$M \ll 3$ 表示 M 的所有位左移三位, 右侧填充 0



$M \ll 8$ 表示 M 的所有位左移八位, 右侧填充 0 (原有 8 个位全部移走, 剩下的全填充为 0)



用代码检查一下上述手工运行的结果是否正确,

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    unsigned char M = 0xA7;
```

```
    unsigned char result;
```

```
    result = M >> 1;
```

```
    printf("%02X \n", result);
```

```
    result = M >> 3;
```

```
    printf("%02X \n", result);
```

```
    result = M >> 8;
```

```
    printf("%02X \n", result);
```

```
    result = M << 1;
```

```
    printf("%02X \n", result);
```

```
    result = M << 3;
```

```
    printf("%02X \n", result);
```

```
    result = M << 8;
```

```
    printf("%02X \n", result);
```

```
    return 0;
```

```
}
```

对于无符号整型(unsigned char / unsigned short / unsigned int)来说, 移位操作是很直观和容易理解的。实际工程应用中, 只对无符号整型进行移位操作。

对有符号整型(char / short / int)来说, 其移位操作的规则比较复杂、且难以理解。幸运的是, 在工程一般不对有符号整数进行移位操作, 因为对有符号数移位不直观, 也没有什么实际意义。负整数在右移时, 高位(符号位)不变, 左侧填充 1。而其他情况下, 均与无符号数的移位规则相同。

6.8.4 应用举例

定义变量 unsigned char flag = 0;

(1) 置 1 操作

将 flag 的 bit5 设置为 1, 其余位保持不变

```
flag |= ( 1u << 5 ); // 注: 1u<<5 等于 0010 0000
```

(2) 清 0 操作, 其余位保持不变

将 flag 的 bit5 清空为 0:

```
flag &= ~(1u << 5); // 注: ~(1u<<5)等于 1101 1111
```

(3) 判断 bit5 是否为 1

```
unsigned char mask = 1u << 5;
if ( flag & mask )
{
    printf("bit5 is set \n");
}
else
{
    printf("bit5 is unset \n");
}
```

6.8.5 例题：把 unsigned int 整数转为 4 个字节

对一个 unsigned int 型变量来说，它占据了 4 个字节。但是如何取得 4 个字节的值呢？可以用移位实现。例如以下的代码中，将变量 a 右移 24 位，则 a 的最高 8 位(bit24-bit31)被移到了低 8 位上，可以直接转成 unsigned char 来保存。依此类推，可以得到 4 个字节的值。

```
unsigned int a = 0x12345678;

unsigned char buf[4];

buf[0] = a >> 24;

buf[1] = a >> 16;

buf[2] = a >> 8;

buf[3] = a;
```

此时，我们得到的结果是: buf[0]: 0x12, buf[1]: 0x34, buf[2]: 0x56, buf[3]: 0x78

6.9 类型的转换与提升

在 char / short / int / double / float 之间，是可以互相转换的。

例如，

```
int a = 12.34;
```

此语句中左侧为 int 型变量，右侧为 double 型的值，结果是 a 的值为 12，小数部分被截断。所以编辑器给出一个警告，提示“从 double 到 int 可能丢失数据”。如下图所示，



如果反过来，把一个 int 型数值赋值给 double 型变量，则不会损失数据。例如下面的代码，

```
double a = 12;
```

此语句中左侧为 double 型，右侧为 int 型，由于 double 的表示范围大于 int，所以不会丢失数据。

以上给事实给了我们一个基本的概念：每一种类型都有一种表示范围，从高到低转换会发生数据损失或截断，从低到高转换不会发生数据损失。它们由低到高依次是 char < short < int < double, float < double。由于 int 和 float 都是 4 个字节，它们之间的转换没那么简单。

在算术表达式中, 当不同的类型混合运算时, 编辑器会根据级别最高的类型先进行提升, 把所有的操作数都操升到最高级别, 然后再进行运算。这样做的目的是避免数据丢失。

例如, 在下面的表达式中, 最高类型为 `double`, 所以编译器自动将各操作数都提升为 `double` 型再进行计算:

```
12 + 78.5 ;
```

以上的转换由编译器自动完成, 称为“隐式转换”。除此之外, 可以使用“显式转换”强制的将类型向上或向下转换。例如,

```
(double) 100 ; // 向上提升为 double, 整个表达式的值类型变成 double
```

`(double) 100/8;` // 将 100 向上提升为 `double`, 致使所有的操作数都向上提升, 最终表达式的值为 12.50;

```
100/8; // 对比一下, 这个表达式是两个 int 作整除, 值为 12
```

```
13 + (int) 25.3; // (int)将 25.3 强制转换成 25
```

6.10 优先级与结合顺序

当一个表达式中出现多个运算符时, 需要考虑优先级的问题。以最简单的算术运算符为例, 它们的优先级从高到低依次为 `* / % + -`。观察下面的代码,

```
int a = 10 + 11 * 12 / 13 - 14 * 15;
```

根据优先级顺序, 高优先级的先结合, 相当于,

```
int a = (10 - (((11 * 12) / 13) + (14 * 15)));
```

还好, 这些 C++ 的乘除加减和数学里的相似, 所以我们凭借有限的数学知识, 可以迅速判断出哪些操作数被先结合。但是情况再复杂一点, 就很难迅速判断了, 因为操作符的种类繁多, 很难记住的优先级高低。

例如,

```
a && !b || c + d > 100 && c < 10    (逻辑表达式混用)
```

```
a + flag & 0x0E + b < 128 ? 1 : 0 (位表达式、条件表达式)
```

6.10.1 使用括号

幸运的是，想要完全记住所有操作符的优先级是不太可能的，这样做也没有必要。原因是，这么复杂的表达式，即使你看的懂，也不能保证别人也看得懂。当一个表达式越复杂时，其可读性就越差。代码的可读性是一件非常重要的事情。

那么怎样才能既不用太多记忆，又保证正确性和可读性呢？答案很简单：**积极使用括号**。

无论何种场景，显式地使用小括号来指定结合顺序，会使你的代码的可读性大增，并且能保证完全正确。当你合理的使用小括号后，别人可以很容易地读懂你的代码，不会有误解和歧义。

比如，我们对下面这种比较复杂的式子使用小括号，

```
a && ( !b ) || ((c + d > 100) && (c < 10))  
a + (flag & 0x0E) + (b<128 ? 1: 0) (位表达式、条件表达式)
```

6.10.2 常用的优先级

实际上，在前面每一节的操作符的表格里，操作符已经是按优先级排列的了。读者应该记住它们的相对优先级就可以了。例如，在算术操作符中，要记住 * / 比 + - 高就够用了，在逻辑操作符里，要记住 ! 高于 && 高于 || 就可以了。

这样就可以少写一些小括号，毕竟如果小括号太多了，那可读性反而会下降。

例如，由于大多程序员都明白 ! && || 这个顺序，所以它们联立时可以不加小括号，

```
a && !b || (c + d > 100) && (c < 10)
```

总结为一条原则：**常见的优先级不加小括号，没有把握的优先级加小括号**。

本书习题：本书配套在线题库（含答案及解析），请登录官网 <http://afanihao.cn> 查看。

本书相关资源，一律到官网获取：

- (1) 视频教程：约 90 集，覆盖全部内容。建议与文字教程结合使用。
- (2) 示例代码：本文字教程的全部示例源码。

7. 语句

本章介绍语句的概念，以及几种常用的几种控制语句 if / switch / for / while。

7.1 什么叫语句

简单地说，以分号结束的一行，称为一条语句（Statement）。

例如，下面的代码包括了 4 条语句：

```
int b = 1;
int a = b;
a = 10;
printf("hello world!\n");
```

然而多个语句也可以写在同一行里：

```
int b = 1; int a = b; a = 10;
```

这样写在语法上没有任何问题，只是可读性较差。这也说明，分号才是划分语句的标志，即使同一行里也可以存在多个语句。

7.1.1 空语句

一个语句的内容可以空，而只有一个分号。空语句的语法地位和普通的一条语句是相同的。

例如，下面的这些只有单个分号的语句称为空语句，

```
; // 空语句
; // 空语句
int a = 10; // 正常语句
```

7.1.2 复合语句

可以将大括号将多个语句又大括号组合起来，称为复合语句。例如，

////////// 例 CH07_A1 //////////

```
#include <stdio.h>

int main()
{
    int a = 10;
    //下面的复合语句里包含了 3 条单语句
    {
        a += 2;
```

```
        a *= 3;

        printf("a=%d \n", a);
    }

    return 0;
}
```

复合语句在语法上和一条单语句的地位是相同的，所以在一条单语句出现的地方，都可以替换为复合语句。

(1) 复合语句内可以含有 0 条语句

也就是说，大括号内可以为空。

例如，

```
int main()
{
    //下面的复合语句里包含了 0 条单语句

    {
    }

    return 0;
}
```

(2) 复合语句内可以嵌套复合语句

复合语句对上一个层次来说，相当于一条单语句。所以，复合语句内还有嵌套复合语句。例如，

```
int main()
{
    int a = 10;
    {
        // 嵌套一个复合语句

        {
            a += 2;
            a *= 3;
        }

        // 嵌套一个复合语句
    }
}
```

```

        {
            a -= 1;
            a /= 2;
        }
    }
    return 0;
}

```

7.2 if 语句

if 语句用于判断一个条件是否成立：当条件成立时，做一件事情；当条件不成立时，做另一件事情。其基本语法形式为：

```

if (expr)
    statement1
else
    statement2

```

语法规则：

当 expr 为真（非 0）时，执行语句 statement1；否则，执行语句 statement2。

其中，statement1 和 statement2 允许是一条单语句、空语句、或复合语句。

例如，用 x 表示一次考试的成绩，当 x 大于或等于 60 分时，提示 Pass，否则提示 Fail。可以用 if 语句来实现，示例代码如下：（为了方便说明，在每行前面加上行号标注，行号部分不属于代码）

```

////////// 例 CH07_B1 //////////
01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可以通过 scanf 让用户输入
05      if(x >= 60)
06          printf("Pass !\n");
07      else
08          printf("Fail !\n");

```

```
09         return 0;
10     }
```

过程分析:

第 1 种情况: 令 $x=61$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 1, 执行[08]

=> 执行[08] : `printf("Pass !\n")`

=> 执行[09] : `return 0`

第 2 种情况: 令 $x=59$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 0, 执行[06]

=> 执行[06] : `printf("Fail !\n")`

=> 执行[09] : `return 0`

7.2.1 使用复合语句

根据 if 语句的语法, 在 if 和 else 后面只可以接一条单语句, 或者一个复合语句。所以, 当需要由多条语句来完成一个处理时, 可以将多个条语句组合成复合语句。例如,

//////////////////// 例 CH07_B2 //////////////////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可能通过 scanf 让用户输入
05      if(x >= 60)
06      {
07          printf("Pass !\n");
08          printf("Go for a celebration! \n"); // 庆祝一下
09      }
10      else
11      {
```

```
12             printf("Fail !\n");
13             printf("Pay more efforts !\n"); // 更须努力
14         }
15     return 0;
16 }
```

记住，绝对不能写成下面这样，这样是有语法错误的。

```
if(x >= 60)
    printf("Pass !\n");
    printf("Go for a celebration! \n"); // 庆祝一下
else
    printf("Fail !\n");
    printf("Pay more efforts !\n"); // 更须努力
```

注：从可读性的角度来说，本书建议读者在 **if** 和 **else** 后面统一加复合语句，即使只有一条语句，也加上大括号。

7.2.2 最简形式 if

我们可以把 **else** 的处理部分省略，只保留 **if** 部分。其语法形式为：

```
if (expr)
    statement1
```

语法规则：当 **expr** 为真（非 0）时，执行语句 **statement1**。

在示例 CH07_A3 中，只对 $x \geq 60$ 的情况做了处理；如果不满足条件，则什么都不做。

////////////////// CH07_B3 //////////////////////

```
01 #include <stdio.h>
02 int main()
03 {
04     int x = 61; // x 可能通过 scanf 让用户输入
05     if(x >= 60)
06     {
07         printf("Pass !\n");
```

```
08             printf("Go for a celebration! \n"); // 庆祝一下
09         }
10         printf("That's over \n");
11         return 0;
12     }
```

过程分析:

第 1 种情况: 令 $x=61$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 1, 执行[07]

=> 执行[07] : `printf("Pass !\n")`

=> 执行[08] : `printf("Go for a celebration! \n")`

=> 执行[10] : `printf("That's over \n")`

=> 执行[11] : `return 0`

第 2 种情况: 令 $x=59$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 0, 执行[10]

=> 执行[10] : `printf("That's over \n")`

=> 执行[11] : `return 0`

7.2.3 完全形式 if...else if ... else if...else

本小节介绍 if 语句的完全形式, 它是对多个条件逐一判断和处理的一种语句。其语法形式为:

```
if (expr1)
    statement1
else if(expr2)
    statement2
else if ...
    ...
else
    statementN
```


语法规则: (多组条件 `expr1, expr2, ...`, 逐个判断, 如果满足条件则执行相应语句)

如果 `expr1` 非 0, 则执行 `statement1`, 然后退出语句;

如果 `expr2` 非 0, 则执行 `statement2`, 然后退出语句;

... ..

如果所有条件均未满足, 则执行最后一个 `else` 语句。

注意:

(1) 如果有一个 `else if` 得到满足, 那么后面就不会接着判断其他条件了。

(2) 最后面的 `else` 语句可以省略, 即 `if ... else if ... else if`, 最后可以没有 `else` 语句

例如, 仍然用 `x` 表示一次考试的成绩, 分为 ABCDE 五档成绩, 位于[90,100]为 A, [80,90]为 B, [70-80]为 C, [60,70]为 D, [0-60]为 E。在示例 CH07_B4 中, 用 `if` 语句实现。

////////// 例 CH07_B4 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可能通过 scanf 让用户输入
05      if(x>=90)
06          printf("Got A \n");
07      else if(x>=80)
08          printf("Got B \n");
09      else if(x>=70)
10          printf("Got C \n");
11      else if(x>=60)
12          printf("Got D \n");
13      else
14          printf("Got E \n");
15      printf("That's over \n");
16      return 0;
17 }
```

过程分析: 令 `x=71`

=> 执行[04] : `int x = 71`

=> 执行[05] : 表达式 `x>=90` 的值为 0, 继续判断其他条件
=> 执行[07] : 表达式 `x>=80` 的值为 0, 继续判断其他条件
=> 执行[09] : 表达式 `x>=80` 的值为 0, 条件满足, 执行此条 `else if` 语句
=> 执行[10] : `printf("Got C \n")`, 然后跳出 `if`, 不再判断其他条件
=> 执行[15] : `printf("That's over \n")`
=> 执行[16] : `return 0`

7.3 switch 语句

这一节介绍 `switch` 语句, 它的作用是根据不同的选项, 跳转到不同的分支处理。其语法形式为:

```
switch(expr)
{
case OPTION_1:
    break;
case OPTION_2:
    break;
case OPTION_....:
    break;
default:
    break;
}
```

其中, `expr`: 表达式的值必须为整型, `OPTION`: 必须为整形常量。

语法规则: 根据 `expr` 的值, 寻找匹配的 `case`。如果某个 `OPTION_X` 与 `expr` 相等, 则跳转到 `OPTION_X` 处执行。如果没有任何匹配的 `case`, 则跳到 `default` 处执行。如果不存在 `default` 标签, 则退出 `switch` 语句。

其语法要素为:

- ① `expr` 必须为整型, 不能是小数或其他类型
- ② `OPTION` 必须是整型常量, 不能是变量。
- ③ `case` 与 `default` 行称为标签, 行尾以冒号结束
- ④ 允许不写 `default` 标签

⑤ 允许不写任何 case 标签

我们来看一个例子，通过这个例子来学习它的用法。

例 CH07_C1：周一去上班，周二去出差，其他待在家。代码如下：

//////////例 CH07_C1 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int day = 2; // 用户输入数字 0-6，代表周日、周一到周六
05      switch(day)
06      {
07          case 1:
08              printf("Go to office\n");
09              break;
10          case 2:
11              printf("Go out \n");
12              break;
13          default:
14              printf("Stay at home \n");
15              break;
16      }
17      return 0;
18  }
```

过程分析：

第 1 种情况：令 day=2

=> 执行[05]：switch(day)，寻找匹配的 case ... 发现第[10]行匹配，跳过去执行...

=> 执行[11]：printf("Go out \n")

=> 执行[12]：break; (跳出 switch 语句)

=> 执行[17]：return 0;

第 2 种情况: 令 `day=3`

=> 执行[05] : `switch(day)`...没有匹配的 `case`...发现 `default`, 跳过去执行...

=> 执行[14] : `printf("Stay at home \n");`

=> 执行[15] : `break;` (跳出 `switch` 语句)

=> 执行[17] : `return 0;`

7.3.1 匹配

标签的匹配过程详细分解为 3 步:

(1) 寻找匹配的 `case`

(2) 如果没有 `case` 匹配, 寻找 `default`

(3) 如果没有 `case` 匹配, 也没有 `default`, 则退出 `switch` 语句

7.3.2 跳转与执行

当发现到匹配的标签后, 直接跳转到该标签的下一行继续执行。注意, 标签本身不算是一行语句。跳过到指定位置后, 程序会一直执行, 直到:

(1) 到达 `switch` 语句的末尾 (大括号的结束), 自然退出 `switch` 语句

(2) 或者遇到 `break` 语句, 则直接退出 `switch` 语句

其中, 第 (2) 条是需要仔细理解的。在关键字 `break` 后直接加上分号, 就是称为 `break` 语句, 其作用是跳出当前层次的 `switch` 语句。

每个分支处理后面都应该加上一条 `break` 语句, 表示分支处理结束。但如果不加 `break` 语句, 也并不是语法错误, 只是运行的结果可能会显得意外, 那么运行时会发生什么呢?

在示例 CH07_C2 中, 每个 `case` 和 `default` 分支处理中都没有加 `break` 语句, 令 `x=1`, 观察运行的结果。

////////// 例 CH07_C2 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int day = 1; // 用户输入数字 0-6, 代表周日、周一到周六
05      switch(day)
06      {
07          case 1:
08              printf("Go to office\n");
```

```
09         case 2:
10             printf("Go out \n");
11         default:
12             printf("Stay at home \n");
13     }
14     return 0;
15 }
```

程序分析：令 day=1

=> 执行[05]：switch(day) ... 发现第[07]行 case 匹配，跳过去执行...

=> 执行[08]：printf("Go to office\n");

=> 执行[10]：printf("Go out \n");

=> 执行[12]：printf("Stay at home \n");

=> 执行[14]：return 0;

是的，虽然语法上没有错误，可是运行结果并不是我们所希望的。我们原本希望当输入 day=1 时，只执行 Go to office。可是由于我们没有加 break 语句，使得程序继续往下执行，连续执行了后面的 Go out 和 Stay at home。

7.3.3 注意事项

在一般情况下，当我们自己书写代码时，按照常规的框架来写，不太可能发生错误。如下面所示意的这样：

```
switch (...)
{
case ...
    break;
case ...
    break;
default:
    break;
};
```

但当我们阅读别人的代码时，如果别人的代码不是这么规范，就得注意了。switch 语句可能出错的语法细节比较多，我们一一核对一下。

- (1) 检查有没有 break 语句，如果没有 break 语句，则程序会继续往下执行的。

(2) 多个标签可以写在一起

```
switch (...)  
{  
  case 1: case 2:  
  case 3: case 4:  
    printf("yes, 1234 is ok \n");  
    break;  
}
```

表示当选项为 1, 2, 3, 4 时都跳转到相同的位置执行

(3) default 标签的位置是自由的

像下面这样写也没问题。

```
switch (...)  
{  
  default:  
    break;  
  case ...  
    break;  
};
```

(3) switch 和 case 的选项值都必须是整型

像下面这样是有语法错误的,

```
switch(1.234)  
{  
}
```

(4) case 的选项值必须是常量

在下面的代码中, case 的值用了变量, 这样是不允许的。

```
int OPTION = 3;  
int choice = 0;
```

```
switch(choice) // OK, 此处可以是变量
{
case OPTION: // 语法错误: case 的值必须是常量
}
```

7.4 for 语句

7.4.1 引例

本小节通过一个例子, 来说明存在什么样问题, 而本节的语法规则是用于解决此类问题的。

引例:

现有一个长度为 100 的 int 数组, 要求初始化各元素的值为 1,2,3,..., 100。我们可以不怕麻烦地写上一百行, 来完成这个要求:

```
int a[100];
a[0] = 1;
a[1] = 2;
... 此处略去几十行 ...
a[99] = 100;
```

对于这样的不胜其烦的工作, 有没有什么好的解决办法呢?

7.4.2 for 语句

使用 for 语句, 可以完成一些可以用循环去完成的、带有规律性的工作。其语法形式为:

```
for ( expr1 ; expr2; expr3)
    statement
```

其中, `expr1,expr2,expr3` 是 3 个表达式, `statement` 是一条单语句或复合语句。

语法规则: (循环如何被执行)

- ① 初始化: 执行 `expr1`。把 `expr1` 称为初始化表达式。
- ② 终止条件: 执行 `expr2`。若 `expr2` 的值为真, 则执行第③步。如果 `expr2` 为假, 则退出 for 语句。
- ③ 循环体: 执行 `statement`
- ④ 后置表达式: 执行 `expr3`
- ⑤ 继续下一轮循环: 跳到第②步

简而言之：每一轮开始之前，都要先判断一个 `expr2`，如果 `expr2` 为假则退出循环。

现在我们使用 `for` 语句来完成引例中的要求。代码如下：

////////// 例 CH07_D1 //////////

```
#include <stdio.h>

int main()
{
    int a[100];
    int i;
    for(i=0; i<100; i++)
    {
        a[i] = i + 1;
    }
    return 0;
}
```

程序分析：

- ① 初始化 `i = 0` （只执行一次）
- ② 判断终止： `i` 为 0， `i<100` 为真，条件满足，继续循环
- ③ 循环体： `a[i] = i + 1`；则 `a[0]` 的值为 1
- ④ 后置语句： `i++` ， `i` 的值变成 1
- ⑤ 判断终止： `i` 为 1， `i<100` 为真，条件满足，继续循环
- ⑥ 循环体： `a[i] = i + 1`；则 `a[1]` 的值为 2
- ⑦ 后置语句： `i++` ， `i` 的值变成 2
- ⑧ 判断终止： 如此反复... ..
- ⑨ 判断终止： : `i` 为 100， `i<100` 为假，退出 `for` 语句

7.4.3 变形 1：省略初始表达式

`for` 语句的变形的一种，就是省去初始表达式。下面的写法和例 CH07_D1 的效果是完全一样的。

////////// 例 CH07_D2 //////////

```
#include <stdio.h>
```



```
int main()
{
    int a[100];
    int i = 0; // 在 for 语句之前面初始化
    for( ; i<100; i++) // for 语句中省去第一个表达式
    {
        a[i] = i + 1;
    }
    return 0;
}
```

7.4.4 变形 2: 省略初始表达式

for 语句的变形的一种, 就是省去第二个表达式, 表示总是执行循环体。通常情况下, 可以把判断终止的语句写在 for 语句里面, 并使用 break 语句跳过 for 循环。

下面的写法和例 CH07_D1 的效果是完全一样的。

////////// 例 CH07_D3 //////////

```
#include <stdio.h>

int main()
{
    int a[100];
    int i;
    for( i=0 ; ; i++) // 省去第 2 个表达式
    {
        if(i>=100)
        {
            break; // break 用于退出 for 语句
        }
        a[i] = i + 1;
    }
    return 0;
}
```

7.4.5 变形 3: 省略后置表达式

for 语句的变形的一种, 就是省去第三个表达式, 取而代之地是, 把后置语句写在循环体的最后面。

```
////////// 例 CH07_D4 //////////
#include <stdio.h>

int main()
{
    int a[100];
    int i;
    for( i=0 ; i<100 ; ) // 省去第 3 个表达式
    {
        a[i] = i + 1;
        i++; // 后置过程写在大括号里面
    }
    return 0;
}
```

7.4.6 变形 4: 全部置空

全部置空也是可以的, 相当于没有初始化, 没有后置语句, 永远执行的循环。

```
for(;;)
{
    printf("... never stop ..... \n");
}
```

由于没有终止条件, 所以上述循环体被无限次地执行下去。

7.4.7 break 语句

如 for 语句中存在 break 语句, 则它的作用是跳出 for 语句, 退出循环。前面已经有演示了。

7.4.8 continue 语句

continue 语句如果存在于 for 语句的大括号内, 当 continue 语句被执行时, 表示结束本轮、直接进入下一轮循环。即, continue 后的语句在本轮被忽略、不被执行。

例: 打印 1,100 之间的偶数, 并统计偶数的个数。

```
////////// CH07_D5 //////////
```

```
#include <stdio.h>

int main()
{
    int count = 0;
    for(int i=1 ; i<= 100; i+=1 )
    {
        if(i % 2)
            continue; // 后面的语句被跳过...

        count ++;
        printf("Even Number: %d \n", i);
    }
    printf("Total : %d \n", count);
    return 0;
}
```

7.5 while 语句

while 语句也用于实现循环，其基本形式为：

```
while(expr)
    statement
```

其中，`expr` 表达式，`statement` 是一条单语句或复合语句。

语法规则：

- ① 执行 `expr`。若 `expr` 的值为真，则执行②。如 `expr` 为假，则退出 `while` 语句。
- ② 执行 `statement`
- ③ 跳到第①步，继续下一轮循环

简而言之：每一轮开始之前，都要先判断一下 `expr` 的值，如果 `expr` 为假则退出循环。

`while` 语句和 `for` 语句本质上没有区别，都是用于实现循环。使用 `for` 语句能够实现的功能，可以很容易地改用 `while` 语句来实现。

例如，可以使用 `while` 语句来完成例 CH07_D1 中的功能，将一个长度为 100 的数组填充上初始值，可以这么实现：

////////// 例 CH07_E1 //////////

```
#include <stdio.h>

int main()
{
    int a[100];

    int i = 0; // 初始化

    while(i<100)
    {
        a[i] = i + 1;

        i++; // 后置过程
    }

    return 0;
}
```

简单地分析一下这个程序:

- ① i 的初始值为 0
- ② 当 i<100 时, 执行循环体。这意味着, 循环体一共被执行了 100 次。

7.5.1 变形: 判断判断内置

`while` 的循环体中也可以存在 `break` 语句和 `continue` 语句, 其意义和 `for` 语句中阐述的完全一样: `break` 语句用于退出循环, `continue` 循环用于跳过本轮、直接进入下一轮循环。

`while` 语句也有一种小小的变形, 那就是条件判断的表达式内置, 放在大括号来判断。在下例中, `while(1)` 表示循环体总是被执行。 `if(i>=100) break` 表示当 `i>=100` 成立时退出循环。

////////// 例 CH07_E2 //////////

```
#include <stdio.h>

int main()
{
    int a[100];

    int i = 0; // 初始化

    while(1) // 总是执行
    {
        if(i>=100) break; // 条件判断放在大括号里面

        a[i] = i + 1;
    }
}
```

```
        i++; // 后置过程
    }
    return 0;
}
```

7.5.2 例题

下面通过一个实例来练习 `while` 语句的用法。

例：要示用户在控制台输入多个整数，并保存到数组中。当用户输入的数字是 0 时，结束输入。最多允许输入 128 个整数。

我们可以用一个长度为 128 的数组来存储用户的输入，并用一个变量 `count` 来计数。每当用户输入一个数，我们把它存入数组中。注意，`count` 的值要初始化为 0。具体的实现代码如下：

```
//////////例 CH07_E3 //////////
#include <stdio.h>

int main()
{
    int numbers[128]; // 最多接收 128 个数
    int count = 0; // 一共接收多少个少数
    while(1)
    {
        printf("输入一个正整数:");
        int ch = -1;
        scanf("%d", &ch);
        if(ch <= 0)
        {
            // 输入数字是 0、或非法输入时，结束输入
            break;
        }
        else
        {
            // 输入数字不是 0，则保存此数字
            numbers[count] = ch;
```

```
        count ++;
    }
}
// 输出这些数
printf("----- result -----\n");
for(int i=0; i<count; i++)
{
    printf("%d ", numbers[i]);
}
}
```

7.6 do...while 语句

除了 for 语句、while 语句之外，还有一种 do...while 语句可用于表示循环。其基本形式为：

```
do
{
    statement
}while(expr);
```

其中，expr 表达式，statement 是一条单语句或复合语句。

语法规则：

① 执行 statement。

②若 expr 的值为真，则执行①。如 expr 为假，则退出 while 语句。

简而言之：先执行一轮，再检测 expr 的值，如果 expr 为真则接着执行下一轮。如果为假则结束循环。

C/C++语言中的三种循环语法 for, while, do...while 本质没有区别，可以互相转换。只是在形式上和 while 语句相比，do...while 语句适合用于“至少执行一次”的循环，它是先执行一次、再决定要不要继续循环的。

7.6.1 例题

假设用户的密码是一个 3 位整数。令用户输入密码，如果输入成功，则提示"Welcome"。如果输入失败，提示"Bad Password!"，并提示其重新输入密码。最多输入 3 次，如果 3 次均未成功，则提示"User Locked"。

我们用 do...while 语句来实现这个功能, 代码如下 (读者可以把它改成 for 或 while 来实现):

```
//////////例 CH07_F1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    int key = 123; // 密码  
    int times = 0; // 尝试次数  
    int passed = 0; // 0: 未通过验证, 1: 已输入密码,通过验证  
    do  
    {  
        times ++; // 尝试的次数  
  
        // 提示用户输入密码  
        int input = 0;  
        printf("Please input your password: ");  
        scanf("%d", &input);  
  
        // 检查密码是否正确  
        if(key == input)  
        {  
            passed = 1;  
            break;  
        }  
        else  
        {  
            printf("Bad Password!\n");  
        }  
    } while (times < 3);
```

```
// 有没有通过验证

if(passed)
{
    printf("Welcome!\n");
}
else
{
    printf("User Locked!\n");
}

return 0;
}
```

7.7 综合例题 1

例：一个班级有 15 人进行了一次考试，他/她们的考试分数用整型数组表示，对分数划分三个 3 等级：大于 80，A 档；大于 60，B 档；其他，C 档。

在例 CH07_G1 中，通过一个 for 循环来遍历数组、访问数组中的每个元素。用 if 语句来判断它的值，如果属于 A 档，则将计数 num_of_A 增加 1，依次类推。代码如下：

//////////例 CH07_G2 //////////

```
#include <stdio.h>

int main()
{
    int score[23] =
    {
        89, 98, 80, 87, 56, 57, 68, 68,
        65, 87, 78, 77, 65, 98, 67
    };

    int num_of_A = 0;
    int num_of_B = 0;
    int num_of_C = 0;
    for(int i=0; i<23; i++)
    {
        int s = score[i];
```



```

    if (s >= 80)
    {
        num_of_A ++;
    }
    else if(s >= 60)
    {
        num_of_B ++;
    }
    else
    {
        num_of_C ++;
    }
}

printf("A: %d, B: %d, C: %d \n",num_of_A, num_of_B, num_of_C );
return 0;
}

```

7.8 综合例题 2

打印如下图所示的图形 ($n \times n$ 个字符, 对角线是*号, 其他为空格)。左图是 10×10 , 右图是 9×9 的字符阵。

以 10×10 为例, $n=10$, 先找找其规律:

第 0 行: 0 列为*, 9 列为*

第 1 行: 1 列为*, 8 列为*

...

第 i 行: i 列为*, $n - 1 - i$ 列为*

参考代码如下:

//////////例 CH07_G2 //////////

```
#include <stdio.h>

int main()
{
    int n = 10;
    for(int i=0; i<n; i++) // i: 行号
    {
        for(int j=0; j<n; j++) // j:列号
        {
            if(j == i || j == n-i-1)
            {
                printf("*");
            }
            else
            {
                printf(" ");
            }
        }
        printf("\n"); // 每行末尾
    }
}
```

本书习题: 本书配套在线题库(含答案及解析), 请登录官网 <http://afanihao.cn> 查看。

本书相关资源, 一律到官网获取:

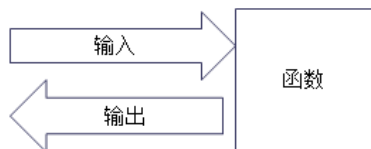
- (1) 视频教程: 约 90 集, 覆盖全部内容。建议与文字教程结合使用。
- (2) 示例代码: 本文字教程的全部示例源码。

8. 函数

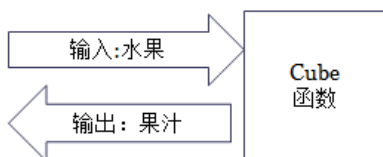
本章介绍函数的概念。函数是一个被包装了的功能模块，送给它一些输入参数，它经过运算得出一个结果并输出返回。

8.1 引例

函数是一个可以完成指定功能的模块，它能执行任务，并把结果返回来。通常我们把函数描述为一个功能盒，送进去一些输入数据，它就产生一些输出数据。

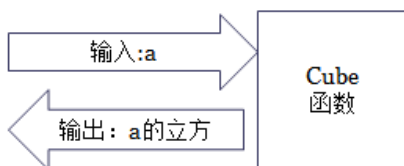


打个比方，榨汁机可以为是一个函数，我们放进去一些水果，让它开始工作，它执行任务，把果汁返回。我们把送进去的“水果”称为输入参数，把函数的结果“果汁”称为返回值。可以用图表示榨汁机的工作过程：



作为使用者，我们通常不会关心函数内部是如何工作的。我们关心的是，它要求输入什么形式的数据，以及它能够返回什么的数据。

再比如，有一个函数，它可以用来求一个数的三次方。这个函数的输入是一个数 a ，输出（返回值）是 a 的三次方。我们把这个函数命名为 **Cube**，表示求三次方。当输入为 2 时，返回值为 8。当输入为 3 时，返回为 27。图示如下：



那么，在 C/C++ 中，函数是如何定义的呢？

8.2 初步认识函数

下面介绍一下函数的定义。其语法形式为：

```
return_type name ( arguments )  
{
```

```
    body
}
```

其中,

name: 函数名。最好起一个有意义的名字。

arguments: 参数列表。每个参数以逗号分开, 如 `int a, int b`。

return_type: 返回值类型。

body: 函数体。即函数的具体实现。

在下面的例子中, 首先定义了一个函数 `cube`, 用于求一个数的三次方。然后在 `main` 函数里使用了这个函数。

```
////////// 例 CH08_A1 //////////
```

```
#include <stdio.h>

int cube (int a)
{
    int result = a * a * a; // 计算出结果
    return result; // 返回结果
}

int main()
{
    int n = cube ( 12 ); // 调用函数
    printf(" result: %d \n", n);
    return 0;
}
```

这里定义一个函数, 它的几个要素为:

函数名: `cube`

参数列表: `int a` , 表示我们应该传入一个 `int` 型的值

返回值: `int` , 表示该函数将返回一个 `int` 型的值

由大括号包围的若干语句, 称为函数体。在函数体中, 至少有一条 `return` 语句, 该语句用于指定函数返回值。也就是说, 我们先计算出结果, 然后用 `return` 语句将这个值返回。

下面讲一下这个函数是如何调用的。

```
int n = cube ( 12 ); // 调用函数
```

使用函数名来调用一个函数，在小括号传入对应参数的值，然后用一个变量 `n` 来保存函数的返回值。

`cube` 函数需要一个 `int` 型的参数，我们传入的是一个 `int` 型的值 12，那么在 `cube` 函数运行时，首先会对参数列里的变量（参变量）进行初始化：

```
int a = 12;
```

然后再进入函数体，执行

```
int result = a * a * a; // 计算出结果
```

```
return result; // 返回结果
```

我们总结一下：

（1）定义函数

- 确定函数的名字
- 确定需要哪些参数
- 确定函数返回值的类型
- 函数体：计算出结果，用 `return` 返回

（2）调用函数

调用函数时，不用关心函数体里的实现，只关心它的名字、参数列表和返回值类型。

8.3 函数的定义

本节具体展示如何定义一个函数。

8.3.1 函数名

函数名的命名规则，就和 C/C++ 里的所有命名规则一样，“字母、数字和下划线的组合，但不能以数字开头”。

除此之外，为了让我们的代码更容易读懂（可读性），我们应该：

（1）命名要有意义

让函数的名字反映其功能，让人可以顾名思义。

例如：

`save_data` 保存数据

`get_input` 获取输入

`average` 求平均值

(2) 规范化命名

通常, 在 C/C++ 有两种函数命名法。本书中两种方式都有使用。

第一种: 全小写, 每个单词以下划线分隔, 如 `save_data`。这是 C 语言里的常见方式。

第二种: 每个单词的首字母大写, 不用下划线。如 `SaveData`。这是 C++ 里的常见方式。

8.3.2 参数列表

参数列表在函数名后面的小括号里指定, 每个参数要有类型、参数名, 每个参数之间用逗号分隔。下面是一些示例:

```
int test (); // 允许不带参数
```

```
int test (int a); // 带一个参数: 参数类型 int, 参数名 a
```

```
int test (int a, double b); // 带一个 int 型参数, 一个 double 型参数
```

在函数的实现中, 那些在函数内部不能确定的量, 就定为参数, 放在参数列表里。

例如, 函数 `Cube` 用于求一个数的立方。算法是不变的, 但输入的那个数是不确定的。因此, 用一个 `int` 型参数表示输入的数。

```
int cube (int n);
```

例如, 求两个数的平方和。自然地, 函数需要外界来指定这两个数, 于是这两个数定为参数。

```
int average(int a, int b);
```

通常一个函数在规划其任务的时候, 它所需要的大部分参数就基本明确了。

8.3.3 返回值类型

要为函数确定一个合适的返回值类型。

例如,

函数 `sum` 用于求两个整数的和。由于两个整数的和仍然是整数, 所以我们可以 `int` 类型作为返回值类型。

```
int sum (int a, int b);
```

函数 `Average` 用于三个整数的平均值。由于三个整数的平均值可能为小数, 所以我们应该用 `double` 型作为返回值类型。

```
double average (int a, int b , int c)
```

8.3.4 函数的实现

大括号里的语句称为函数的实现，又叫函数体。

通常，我们先计算出结果，然后有一个 `return` 语句将结果返回给调用者。

例如，

```
////////// 例 CH08_B1 //////////
```

```
double Average (int a, int b , int c)
{
    double result = (a + b + c) / 3.0; // 计算出结果
    return result; // 用 return 语句返回
}
```

注意，有些初学者以为 `printf` 打印出来的就是返回值。这是错误的，一定要记住返回值必须用 `return` 语句，跟 `printf` 没有关系。

```
double Average (int a, int b , int c)
{
    double result = (a + b + c) / 3.0; // 计算出结果
    printf("%d", result); // 初学者易犯的错误：不要 printf，请用 return 语句
}
```

8.3.5 return 语句的用法

`return` 语句的作用：

- (1) 返回一个值
- (2) 函数立即退出

当 `return` 语句被执行后，无论 `return` 语句处在什么层次的复合语句内，函数立即退出。函数内的任何语句都不再执行。

在示例 CH08_B2 中，`printf("bbbb\n")` 不会被执行。

```
////////// 例 CH08_B2 //////////
```

```
int test()
{
```

```
printf("aaaaa\n");  
return 0; // 此处函数退出  
printf("bbbbb\n"); // 不被执行  
return 1; // 不被执行  
}
```

这意味着，**return** 语句不必一定得写在函数体的最下面一行。实际上，只要我们已经得到函数的结果，就可以立即用 **return** 返回结果并退出函数。

例：判断一个字母是大小还是小写。如果是大写字母，返回 1。如果是小写字母，返回-1。如果根本不是字母，返回 0。在例 CH08_B3 中，定义了函数 **test**，并在 **main** 函数里调用它。

```
////////// CH08_B3 //////////  
#include <stdio.h>  
int test(char ch)  
{  
    if(ch >= 'A' && ch <= 'Z')  
    {  
        return 1;  
    }  
    else if(ch >= 'a' && ch <= 'z')  
    {  
        return -1;  
    }  
    return 0;  
}  
int main()  
{  
    int r = test ('M');  
    printf("result: %d \n", r);  
    return 0;  
}
```


8.4 函数的调用

在 C/C++里, 把函数的使用称为“调用”(Call, Invoke)。在调用一个函数时, 必须知道这个函数的名称、参数的个数及类型、返回值类型。

8.4.1 函数的调用过程

我们用前面的求立方的例子, 来说明函数的调用过程。

////////// 例 CH08_C1 //////////

```
01  #include <stdio.h>
02  int Cube (int a)
03  {
04      int result = a * a * a; // 计算出结果
05      return result; // 返回结果
06  }
07  int main()
08  {
09      int n = Cube ( 4 ); // 调用函数: 传入参数, 得到返回值
10      printf(" result: %d \n", n);
11      return 0;
12  }
```

在 C/C++中, 程序总是从 main 函数开始运行, 于是程序是从第[09]行开始的:

=> 执行[09] : 调用 Cube (4)

=> 执行[03] : 进入函数 Cube, 参变量被初始化 int a = 4

=> 执行[04] : 计算 result 为 64

=> 执行[05] : return 被执行, 程序退出。

=> 执行[09] : 回到原先函数调用的地方, Cube 函数返回值 64 被赋值给了 n

=> 执行[10] : printf 打印出 result 的值

=> 执行[11] : return 被执行, main 函数返回。当 main 函数返回时, 意味着整个程序退出。

8.4.2 注意参数的顺序

在传递参数值给函数时, 要注意参数的顺序。

例如，一个函数用于求一个数的 n 次方式，其函数定义为：

////////// 例 CH08_C2 //////////

```
int Power(int base, int n)
{
    int result = 1;
    for(int i=0; i<n; i++)
    {
        result *= base;
    }
    return result;
}
```

这个函数要求第一个参数是基数的值，第二个参数是幂的值。当我们求 2 的 5 次方时，应该 `Power(2, 5)`，而不是 `Power(5, 2)`。要注意这个顺序问题。

8.4.3 函数的传值调用

这小节讲阐述一个重要的概念：传值调用。

先看一个例子。在例 CH08_C3 中，`Test(n)` 的调用会不会修改 `n` 的值呢？有人会认为 `n` 的值变成了 1001，有人会认为 `n` 的值保持不变。

////////// 例 CH08_C3 //////////

```
#include <stdio.h>

void Test(int a)
{
    a += 1000;
}

int main()
{
    int n = 1;
    Test(n);
    printf("Now: n=%d \n", n); // n 的值是 1001 吗？
    return 0;
}
```

经过测试,我们发现无论在 Test 函数里怎么操作, main 函数里的变量 n 的值始终不变。所以我们有以下结论。

(1) 参变量 a 和原变量 n 是两个不同的变量

Test 函数里的参变量 a, 和 main 函数里的 n, 是两个不同的变量, 对应于不同的内存地址。所以, 在 Test 函数里对 a 的修改是不会影响到 n 的值的。

我们知道, 一个变量的地址用&操作符可以取得, 并在第 3 章中已经学会了如何打印一个变量的地址。于是我们用下面的代码检验一下, 看 a 和 n 这两个变量的地址是否相同。

```
//////////例 CH08_C4 ////////////  
  
#include <stdio.h>  
  
void Test(int a)  
{  
    a += 1000;  
    printf("a 的地址: %08X \n", &a);  
}  
  
int main()  
{  
    int n = 1;  
    printf("n 的地址: %08X \n", &n);  
    Test(n);  
    printf("Now: n=%d \n", n); // n 的值是 1001 吗?  
    return 0;  
}
```

经过测试, 可以发现 a 和 n 的地址不同, 从而证明了 a 和 n 是不同变量。

(2) 传的是变量的值

实际上, 函数的调用过程是传值调用。Test(n)所表示的意思是, “把 n 的值传给函数”, 而不是说把 n 这个变量传给函数。

在前面“函数的调用过程”这一节里, 我们已经展示了参变量的初始化。也就是说, 在进入函数时相当于执行了一句这样的操作:

```
int a = n; // 把 n 的值赋给参变量 a
```

8.4.4 忽略返回值

当调用一个函数时，虽然这个函数有返回值，但是可以忽略这个返回值。也就是说，不用它的返回值。

例如，

//////////例 CH08_C5 //////////

```
#include <stdio.h>

int Test(int a)
{
    printf("this is a test \n");
    return a* a;
}

int main()
{
    Test ( 2 ); // 忽略 Test 函数的返回值
    return 0;
}
```

8.4.5 直接使用返回值

不需要定义一个变量来接收返回值，可以直接使用它。

//////////例 CH08_C6 //////////

```
#include <stdio.h>

int Test(int a)
{
    return a* a;
}

int main()
{
    printf("Ressult: %d \n", Test ( 2 ) ); // 忽略 Test 函数的返回值
    return 0;
}
```

8.5 全局变量和局部变量

在函数内定义的变量，叫做局部变量（Local Variable）。（包含参变量）

在函数外定义的变量，叫做全局变量（Global Variable）。

我们来看一个全局变量的例子。在例 CH08_D1 中，变量 `number` 定义在函数体之外，它是一个全局变量。代码如下所示：

```
////////// 例 CH08_D1 //////////  
  
#include <stdio.h>  
  
int number = 0; // 定义在函数之外，是全局变量  
  
void Add(int n)  
{  
    number += n;  
}  
  
int main()  
{  
    number = 10;  
    Add(1);  
    printf("now: %d \n", number);  
    return 0;  
}
```

全局变量常用于存储一些全局性的数据，它在各个函数中均可以访问。比如，在上例中，在 `main` 函数可以访问 `number`，将 `number` 赋值为 10。在 `Add` 函数也可以访问 `number`，将 `number` 的值增加 `n`。

如果某些功能用局部变量不方便完成，可以考虑使用全部变量。

例如，我们要求一个数组的所有元素的平均值。然而，我们并不知道如何将一个数组内的值全部传递给函数，这时候我们可以用全局变量来实现。在例 CH08_D2 中，将数组定义在函数之外，以便在 `Average` 函数中可以直接访问其值。

```
////////// 例 CH08_D2 //////////  
  
#include <stdio.h>  
  
// 定义一个全局变量  
  
int data[8] = {1,2,3,4,5,6,7,8};
```

```
// 不必传入参数

double Average()
{
    // 在函数里直接读取全局变量 data

    int total = 0;
    for(int i=0; i<8; i++)
    {
        total += data[i];
    }

    return total / 8.0;
}

int main()
{
    double result = Average();
    printf("result: %.3lf \n", result);
    return 0;
}
```

什么时候用局部变量？什么时候用全部变量？原则很简单：如果能用局部变量完成，那就不要用全局变量。

8.6 变量的作用域与生命期

8.6.1 变量的作用域

每一个变量，都有一个有效范围，称为“作用域”。在这个范围之内，这个变量是可以访问的。

局部变量的作用域：

- (1) 从定义之处起生效
- (2) 至大括号结束后失效 (该变量所在的大括号)

在下例中，第一个 `printf` 行中可以访问变量 `b`，因为此处还在变量 `b` 的作用范围内。而第二个 `printf` 行就已对超出一变量 `b` 的作用域，因而编译器报错。

```
#include <stdio.h>

int main()
```

```
{
    if(1)
    {
        int b = 10; // b 生效

        printf("b: %d\n", b); // 可以访问 b
    } // b 失效

    printf("b: %d\n", b); // 编译错误!

    return 0;
}
```

8.6.2 变量的生命期

变量的生命期，是指在程序运行的时候，变量存在的时间。“生命期”这个概念，实质上 和“作用域”说的是同一件事。作用域是从代码角度来说问题，而生命期则是从运行时的角度 来说问题。

(1) 全局变量的生命期是有恒的

在程序运行期间，该全局变量始终存在，始终可以访问。

(2) 局部变量的生命期是短暂的

局部变量的生命期和其作用域是一致的。自定义之处，变量的生命期开始，变量是有效的。 当超出作用域后，该变量生命期结束，该变量失效、不再可以访问。

也就是说，局部变量超出它所在的大括号，其生命期就结束了。

8.7 变量名重名问题

这一节介绍变量名的重名问题。为了便于进一步说明作用域和生命期的相关原理，按照大 括号的层次进行分级。将全局变量定位 0 级。如下所示：

```
int test()
{
    int a = 0; // 第 1 层级

    if(1)
    {
        int b = 0; // 第 2 层级

        while(1)
        {
```

```
        int c = 0; // 第 3 层级
    }
}

return 0;
}
```

如果代码已经使用规范的缩进的话，那么缩进的位置刚好对应了这个变量的层级。

(1) 不同函数内的变量，允许重名

如果两个变量在不同的函数里，那肯定是可以重名的，两者之间不会有影响。

在例 CH08_E1 中，Test 函数中的变量 a 和 b 和 main 函数中的同名变量 a,b 是完全没有影响的。

```
////////// CH08_E1 //////////
#include <stdio.h>

int Test(int a)
{
    int b = a + 10;
    int c = a + b;
    return c;
}

int main()
{
    int a = 10;
    int b = Test(a);
    return 0;
}
```

(2) 可以和上一层次的变量重名

一个变量可以和它上面层次的变量重名。当重名时，优先找本层级的变量。

```
////////// 例 CH08_E2 //////////
#include <stdio.h>

int main()
{
```



```
int a = 1;
if(1)
{
    int a = 2; // 定义一个同名的变量
    printf("level2: a=%d \n", a); // 访问的是本层级的变量 a, 打印值为 2
}
printf("end: a= %d\n", a); // 打印值为 1
return 0;
}
```

(3) 就近原则

当重名发生时, 实际被访问的变量是跟本层次最近的那个变量。在下例中, 不同的层次和有一个变量 `a`, 那实际被访问的变量是离它最近的变量。所以输入值为 101。

////////// 例 CH08_E3 //////////

```
#include <stdio.h>
int a = 100;
int main()
{
    int a = 101;
    if(1)
    {
        printf("a=%d \n", a); // 最近的变量 a 是 101
    }
    return 0;
}
```

注: 显然地, 只在两个变量处于同一层级时, 才不允许重名。

8.8 函数声明与函数定义

当一个 `cpp` 文件里有多个函数, 且它们之间有调用关系时, 那么它们的先后顺序就变得复杂。我们需要保证, 一个函数调用之前, 该函数已经被定义。

例如, 现在的任务是打印所有的字母的 ASCII 码的值。我们可以通过几个函数来实现。在例 CH08_E1 中, 我们定义函数 `is_alpha` 用于判断一个字符是否为字母, 在函数 `print_ascii` 又调用了 `is_alpha` 函数。

```
////////// 例 CH08_F1 //////////

#include <stdio.h>

// 如果 ch 是字母, 则返回 1; 否则返回 0

int is_alpha(char ch)
{
    if(ch >= 'a' && ch <= 'z')
        return 1;

    if(ch >= 'A' && ch <= 'Z')
        return 1;

    return 0;
}

// 打印所有字母的 ASCII 码

void print_ascii()
{
    for(int i=0; i<127; i++)
    {
        if(is_alpha(i))
        {
            printf("%c - %d \n", i, i);
        }
    }
}

int main()
{
    print_ascii();

    return 0;
}
```

总体来说, 这个 `cpp` 文件中的三个函数呈如下调用关系

```
main() -> print_ascii() -> is_alpha()
```

因而它们在书写顺序上要求保证 `is_alpha` 在最前面, `print_ascii` 在中间。可以想象, 当函数越来越多时, 要保证顺序就会变得麻烦。

8.8.1 函数的声明

函数的声明 (Declaration), 在形式上就是把函数定义中的函数体去掉, 只保留函数名、参数列表、返回值类型。并以分号结束。例如,

```
int is_alpha(char ch);
```

```
void print_ascii();
```

此外, 还有一个术语: 函数原型 (Prototype), 也是指函数名、参数列表和返回值类型这三个要素。不过函数原型通常用于日常沟通和文档描述中, 并不是一个语法成分。

当存在一个函数被声明后, 它就可以被直接调用, 而不一定要把函数定义也放在前面。这就解决了前面说的函数调用顺序问题。我们通常把各个函数的声明先写在前面, 然后函数定义就不需要保证顺序了。

例如, 我们对例 CH08_F1 采用函数声明的写法,

```
//////////例 CH08_F2 //////////
```

```
#include <stdio.h>
```

```
// 声明本页面内的函数
```

```
int is_alpha(char ch);
```

```
void print_ascii();
```

```
int main()
```

```
{
```

```
    print_ascii();
```

```
    return 0;
```

```
}
```

```
// 打印所有字母的 ASCII 码
```

```
void print_ascii()
```

```
{
```

```
    for(int i=0; i<127; i++)
```

```
    {
```

```
        if(is_alpha(i))
```

```
        {
```

```
        }
```

```
        printf("%c - %d \n", i, i);
    }
}
```

// 如果 ch 是字母, 则返回 1; 否则返回 0

```
int is_alpha(char ch)
{
    if(ch >= 'a' && ch <= 'z')
        return 1;

    if(ch >= 'A' && ch <= 'Z')
        return 1;

    return 0;
}
```

8.8.2 函数声明相关问题

(1) 在函数声明中, 参数名是可以省略的

例如,

```
int is_alpha( char ); // 只需要指定参数类型, 不需要参数名字
```

(2) 函数声明中的参数名, 和函数定义时的参数名可以不同

例如:

```
void test( char  a); // 声明里的参数名无关紧要
void test( char  m)
{
}
}
```

(3) 函数声明的作用

函数声明, 作用是向编译器声明, 存在这么一个函数, 名字是什么, 参数类型是什么, 返回值是什么。这样编译器便心中有数, 可以为我们检查函数调用的正确性。

8.9 main 函数

main 函数是一个特殊的函数。无论程序中有多少个函数，第一个被执行的函数总是 main 函数。换句话说，main 函数是程序的入口。

main 函数的原型为： `int main()` 。

8.10 参数的隐式转换

一般来说，我们要求传递的值的类型，和函数要求的参数类型应该是一致。

例如，在例 CH08_G1 中，Power 函数要求传入的两个参数为 `int, int` 类型。在调用的时候，`Power(2,5)`，传入的是实际类型是 `int, int`，因而参数类型完全匹配，没有问题。

```
////////// CH08_G1 //////////  
  
#include <stdio.h>  
  
int Power(int base, int n)  
{  
    int result = 1;  
    for(int i=0; i<n; i++)  
    {  
        result *= base;  
    }  
    return result;  
}  
  
int main()  
{  
    int result = Power(2, 5);  
    printf("result: %d \n", result);  
    return 0;  
}
```

我们考虑，如果参数类型不匹配，是不是就一定不能调用了呢？比如，我们调用 `Power(2.1, 5)`，试图求 2.1 的 5 次方。此时，传入的参数值的类型为 `(double, int)`，不匹配，但编译器还是允许通过。

其原理是这样的：当参数类型不匹配时，编译器尝试对参数进行隐式转换。如果允许隐式转换，则编译通过。如果不能隐式转换，则编译不通过。

当我们调用 `Power(2.1, 5)` 时, 编译器相当于做了如下尝试:

```
int base = 2.1; // 参数不匹配, 但支持隐式转换, base 的值为 2 (小数部分被截断)
```

```
int n = 5; // 参数匹配
```

所以, 编译器是允许其编译通过的, 只是报了一个警告 (数据被截断)。我们实际运行的结果也显示, `Power(2.1, 5)` 和 `Power(2, 5)` 的结果都是 32, 这是因为参数的值被截断为整数了。

8.11 *函数名重载

(*初学者可以跳过本节)

在 C++ 里, 允许两个函数的名字相同, 称为函数名重载。也就是说, 在 C++ 里只有名字相同、参数列表也相同 (参数个数, 参数类型) 也相同的函数, 也被认定为重复。

例如, 以下两个函数是不同的函数:

```
double find_max(double a, double b); // 求两个数中的较大值
```

```
double find_max(double a, double b, double c); // 求三个数中的最大值
```

当重载函数被调用时, 编译器会根据参数的个数和类型来匹配不同的函数。在示例 CH08_C1 中, 重载了两个名为 `find_max` 的函数, 并在 `main` 函数中调用。

```
////////// 例 CH08_H1 //////////
```

```
#include <stdio.h>
```

```
// 求两个数中的较大值
```

```
double find_max(double a, double b)
```

```
{
```

```
    return a > b ? a : b;
```

```
}
```

```
// 求三个数中的最大值
```

```
double find_max(double a, double b, double c)
```

```
{
```

```
    double m = a > b ? a : b;
```

```
    return m > c ? m : c;
```

```
}
```

```
int main()
```

```
{
    double m1 = find_max(1.1, 2.2);
    double m2 = find_max(1.1, 2.2, 3.3);
    return 0;
}
```

注：在比较函数是否相同时，忽略其返回值。例如，以下两个函数是否重复呢？

```
double Test (double a, double b)
```

```
{
}
```

```
int Test ( double m, double n)
```

```
{
}
```

根据规则，在比较时只比较函数名称和参数列表，由它们的名称都是 `Test`，参数都是 `(double, double)`，所以这两个函数是重复的，编译器会报错。返回值类型不参与比较。

8.12 *重载函数的匹配

当函数名被重载后，函数的匹配过程过程：首先寻找能精确匹配的函数；如果未能精确匹配，则尝试找一个可以模糊匹配的函数（隐式转换）。

- ① 精确匹配：参数个数相同，类型相同
- ② 模糊匹配：参数个数相同，类型不同、但支持隐式转换

8.12.1 精确匹配

在例 CH08_J1 中，调用 `find_max(1,2)`，其参数值的类型为 `(int, int)`，因而精确匹配到第 2 个 `find_max(int, int)`。

```
//////////例 CH08_J1 //////////
#include <stdio.h>

// 求两个 double 数中的较大值
double find_max(double a, double b)
{
    return a > b ? a : b;
}
```

```
}  
// 求两个 int 数中的较大值  
int find_max(int a, int b)  
{  
    return a > b ? a : b;  
}  
int main()  
{  
    double m1 = find_max(1,2); // 参数值的类型(int, int)  
    return 0;  
}
```

8.12.2 模糊匹配

示例 CH08_J2 展示了没有精确匹配, 但可以模糊匹配的情形。实际输入的参数值为(int, int), 而备选的函数为 find_max(double, double)。虽然未能精确匹配, 但是支持对参数进行隐式转换, 所以最终匹配成功。

```
////////// 例 CH08_J2 //////////  
#include <stdio.h>  
// 求两个数中的较大值  
double find_max(double a, double b)  
{  
    return a > b ? a : b;  
}  
// 求三个数中的最大值  
double find_max(double a, double b, double c)  
{  
    double m = a > b ? a : b;  
    return m > c ? m : c;  
}  
int main()  
{  
    double m1 = find_max(1, 2);
```



```

    return 0;
}

```

下面的例子 CH08_J3 则展示了, 有多个备选函数, 均支持隐式转换的情形。实际输入的参数值为(int, int), 没有精确匹配的函数, 但存在 find_max(double, double)和 find_max(float, float)均支持隐式转换。在这种情况下, 编译器无法自己决定使用哪一个函数, 编译器报错。

////////// 例 CH08_J3 //////////

```

#include <stdio.h>

// 求两个数中的较大值

double find_max(double a, double b)
{
    return a > b ? a : b;
}

// 求两个数中的较大值

float find_max(float a, float b)
{
    return a > b ? a : b;
}

int main()
{
    int m1 = find_max(1, 2);
    return 0;
}

```

通常我们应该对参数进行显式类型转换, 以明确使用哪个版本的重载函数。例如, find_max((double) 1, (double) 2)则能精确匹配到 find_max(double, double)这个函数。

注: 在原来的 C 语言中, 不允许函数名称重复。具体请参考本书附录《C++和 C 的区别》。

8.13 *参数的默认值

(*初学者可以跳过本节)

在 C++中, 引入了参数默认值的语法。此语法在 C 语言中不支持。

在函数的声明或定义之处, 将参变量指其默认值。在函数调用时, 如果指定了这个参数, 则使用实际传入的参数值。或未指定, 则使用默认值。

在例 CH08_K1 中, 函数 Show 的第 3 个参数具有默认值 1。所以, 调用 Show(400,300)时没有指定第 3 个参数, 则第 3 个参数取默认值 1。

////////// 例 CH08_K1 //////////

```
#include <stdio.h>
```

```
void Show ( int x, int y, int z=1) // 第 3 个参数具有默认值
```

```
{
```

```
    printf("location: (%d, %d), layer: %d \n", x, y, z);
```

```
}
```

```
int main()
```

```
{
```

```
    Show(400,300); // 未指定第 3 个参数, 则第 3 个参数取默认值 1
```

```
    Show(200,300, 2); // 指定第 3 个参数
```

```
    return 0;
```

```
}
```

相关的注意事项:

(1) 具有默认值的参数必须列在后面。

例如,

```
void Show ( int x, int y=1, int z=1) // 可以
```

```
void Show ( int x=1, int y, int z) // 错误! 带默认值的参数应该放在后面!
```

(2) 当函数的声明与定义分开时, 应该把默认值写在声明里, 不能写在定义里

////////// 例 CH08_K2 //////////

```
#include <stdio.h>
```

```
void Show ( int x, int y, int z=1); // 函数声明里加默认值
```

```
int main()
```

```
{
```

```
    Show(400,300); // 未指定第 3 个参数, 则第 3 个参数取默认值 1
```

```
    Show(200,300, 2); // 指定第 3 个参数
```

```
    return 0;
```

```
}
```

```
void Show ( int x, int y, int z) // 函数定义之处: 不能加默认值
{
    printf("location: (%d, %d), layer: %d \n", x, y, z);
}
```

8.14 *内联函数

(*初学者请跳过本节。本节地位: 不重要, 可以忽略)

将函数定义前添加一个 `inline` 关键字, 则该函数称为内联的。将函数声明为内联的, 有什么好处呢?

对于以下函数,

```
int max(int a, int b)
{
    return a > b ? a: b;
}
```

我们知道, 当调用函数时要做一些基本的入栈、出栈操作, 这些操作由编译器自己完成、程序员无法看见, 我们将它称为函数调用的基本开销。这就是说, 函数调用本身是有成本的, 即使一个函数的函数体为空, 但在调用它的时候仍然耗费了一定的 CPU。

假设这个这些基本开销等价于 N 行指令, 函数体编译条件到 M 行指令; 如果你的函数体太短, 就可能会发生 $M < N$ 的情形。这相当于开了一列火车, 火车上只坐了一个旅客, 这是效率比较低的事情。

避免比较这种低效率事情的发生, 当发现函数体较短时, 将函数声明为 `inline`, 此时编译器会做特定的优化: 不使用函数调用机制, 而是把代码逻辑直接“内联”到调用点上。这使得代码的运行效率提高, 因为省去了函数调用机制。所以 `inline` 函数是形式上函数, 但并不是按函数机制来调用的。

下面的代码中, 将函数 `max` 修饰为 `inline` 的,

```
inline int max(int a, int b)
{
    return a > b ? a: b;
}

void main()
{
    int a = max (10, 13); // 函数调用的规则没变
```

```
}
```

至于什么情况叫“短”，什么情况叫“长”，这是由编译器自己衡量的。即使你将函数声明为了 `inline`，编译器也不一定会按 `inline` 编译。如果它发现你的代码实际上是很“长”的，它还是会按函数调用的方式来编译。

8.15 *函数的递归调用

(*初学者可以跳过本节)

在 C++ 中，一个函数在定义后可以被其他函数调用，而 `main` 函数则是第一个被执行的函数。在普通情况下，函数的调用关系可能是这样子的：

```
main() -> A() -> B() -> C()
```

实际上，还有一种特殊函数调用情形，示意如下：

互相调用： `main() -> A() -> B() -> A() ...`

调用自己： `main() -> A() -> A() ..`

当一个函数直接或者间接地调用到了自己时，称为递归调用。

递归调用常用于实现特定的算法，它可以简化算法的实现。其主要思路时，缩减问题的规模，将一个高阶的问题转化为低阶相同问题的调用。

例：已知有一列数：1, 1, 2, 3, 5, 8, 13, ... （数学中的斐波那契数列）

其规律是： $F_1=1$, $F_2=1$, $F_n=F(n-1)+F(n-2)$ ，其中 F_n 为第 n 个数。要求写一个函数，来求第 n 项的值 F_n 。

```
////////// 例 CH08_L1 //////////
```

```
#include <stdio.h>
```

```
int Fn(int n)
```

```
{
```

```
    // 终止条件
```

```
    if(n==1)
```

```
        return 1;
```

```
    if(n==2)
```

```
        return 1;
```

```
    // 缩减问题规模
```

```
        return Fn(n-1) + Fn(n-2);
    }

int main()
{
    for(int i=1; i<= 10; i++)
    {
        printf("%d ", Fn(i));
    }
    return 0;
}
```

其中，求 F_n 是一个 N 阶问题，在算法中转化为对第 $N-1$ 阶和 $N-2$ 阶相同的问题的调用。

递归算法的特点：

(1) 将高阶问题降为低阶相同问题

如， `return Fn(n-1) + Fn(n-2);`

(2) 必须设置终止条件，避免无限制递归

如， `if(n==1) return 1;` 当 n 为 1 是最低阶，结束递归

(3) 可以替换为非递归算法，改用循环语法来实现

所有的递归算法，都可以改用 `for` 或 `while` 的语法来实现。

在示例 CH08_L2 中，展示了如何用非递归算法来求得斐波那契数列的第 n 项的值

////////// 例 CH08_L2 //////////

```
int Fn (int n)
{
    if(n==1) return 1;
    if(n==2) return 1;

    int an_2 = 1;
    int an_1 = 1;
    int an = 0;
```

```
for(int i=3; i<=n ; i++)
{
    an = an_2 + an_1; // A(n) = A(n-2) + A(n-1)
    an_2 = an_1;
    an_1 = an;
}
return an;
}
```

本书习题：本书配套在线题库（含答案及解析），请登录官网 <http://afanihao.cn> 查看。

本书相关资源，一律到官网获取：

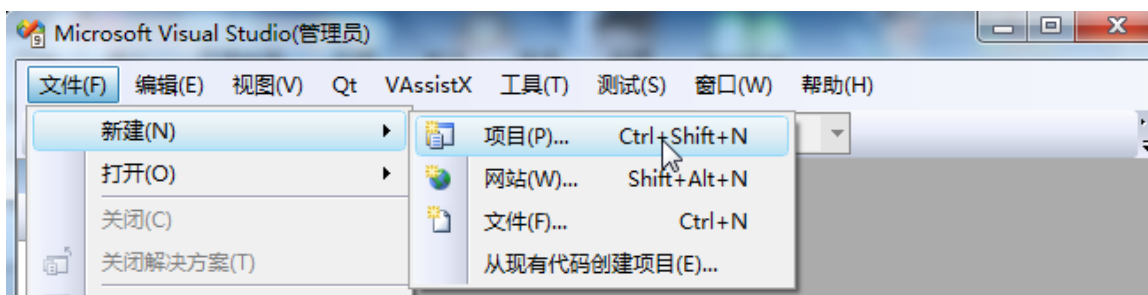
- （1）视频教程：约 90 集，覆盖全部内容。建议与文字教程结合使用。
- （2）示例代码：本文字教程的全部示例源码。

9. 附录用 VC2008 创建项目

本篇介绍的是，用 Visual Studio 2008 里的 Visual C++ 来建立控制台项目。由于 Visual Studio 的各个版本（2005, 2008, 2010, 2012 等）界面和功能类似，所以本篇也适用其他版本。如果你的版本是 Visual Studio 6.0，请到本书的官方网站 <http://afanihao.cn> 上寻找帮助。

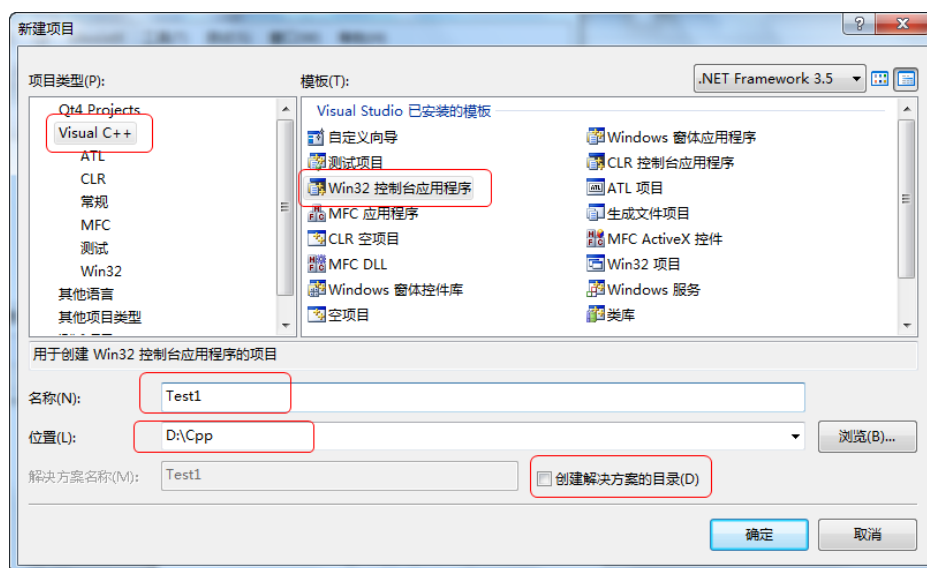
9.1 新建 C++ 项目

选择菜单“文件 | 新建 | 项目”，如下图所示，



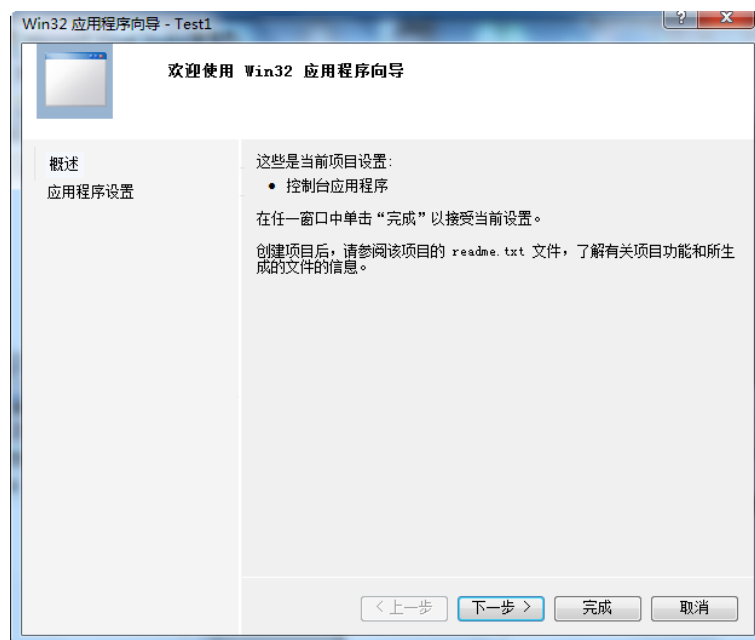
选择菜单项

进入“新建项目对话框”，如下图所示，



新建项目对话框

在新建项目对话框里：① 左侧“项目类型”里选择“Visual C++”② 右侧“模板”里选择“Win32 控制台应用程序”③ 名称输入“Test1”，或自拟一个项目名称④ 选择项目存储的位置“D:\Cpp”目录，或自建一个目录⑤ 不要勾选“创建解决方案的目录”。然后点“确定”按钮，进入“概述”页面，如下图所示，



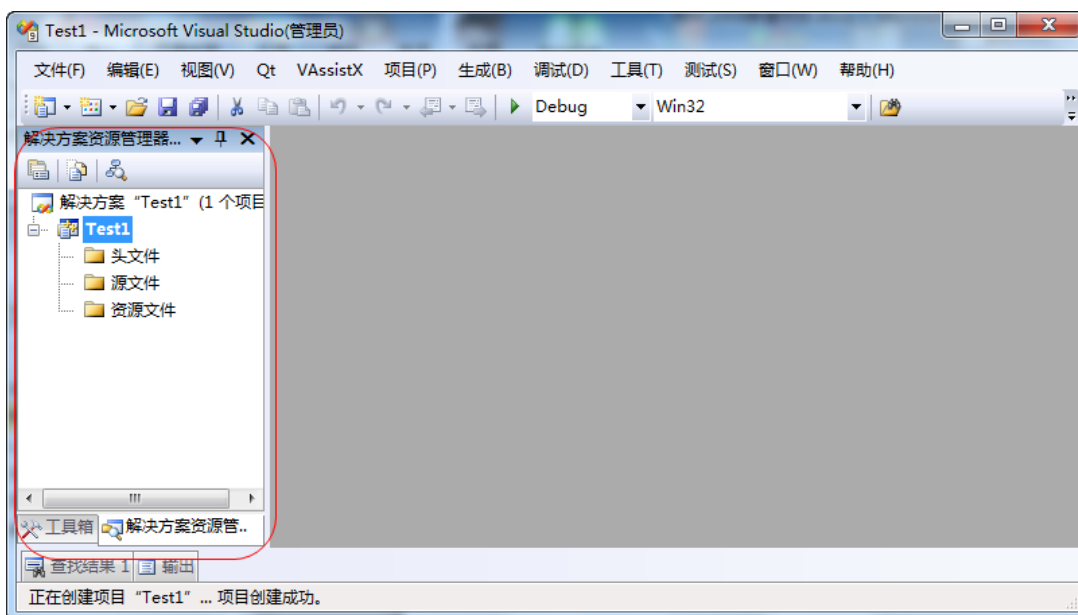
概述页面

在“概述”页面，点“下一步”，进入“应用程序设置”页面，如下图所示



应用程序设置页面

在“应用程序设置”页面里，上面的“应用程序类型”选择为“控制台应用程序”，在下面的“附加选项”里选择为“空项目”。点“完成”按钮，则项目创建完毕。如下图所示。

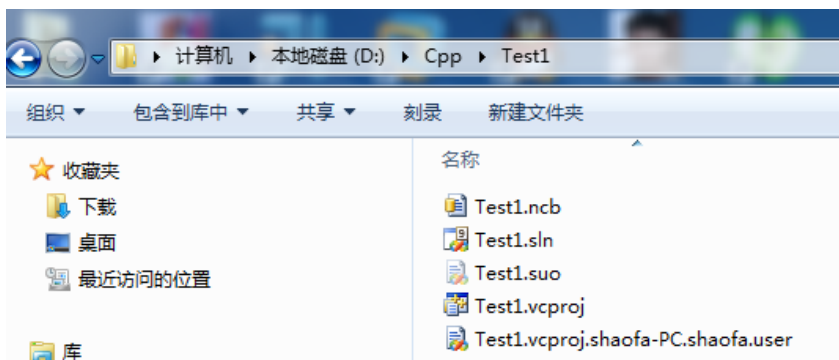


项目界面

此时项目界面的左侧，显示为“解决方案资源管理器”，这里以树状目录的形式显示了项目中的所含的文件列表。现在还没有任何文件。

9.2 查看项目文件夹

在 Windows 的“我的电脑”里，打开项目文件的存储位置：D:\Cpp\Test1，查看项目中的文件，如下图所示，



项目文件夹

也可以用另一种方法打开项目文件夹：在 Visual Studio 2008 的资源管理器里，右键点击 Test1，在快捷菜单里选中“在 Windows 资源管理器里打开文件夹”，也可以打开项目文件目录。

9.3 添加文件到项目

初始建立的项目是“空项目”，项目中不含任何文件。一个 C++ 项目可以含有多个*.cpp 文件和*.h 文件。本节演示，如果在项目中新建*.cpp 文件和*.h 文件。

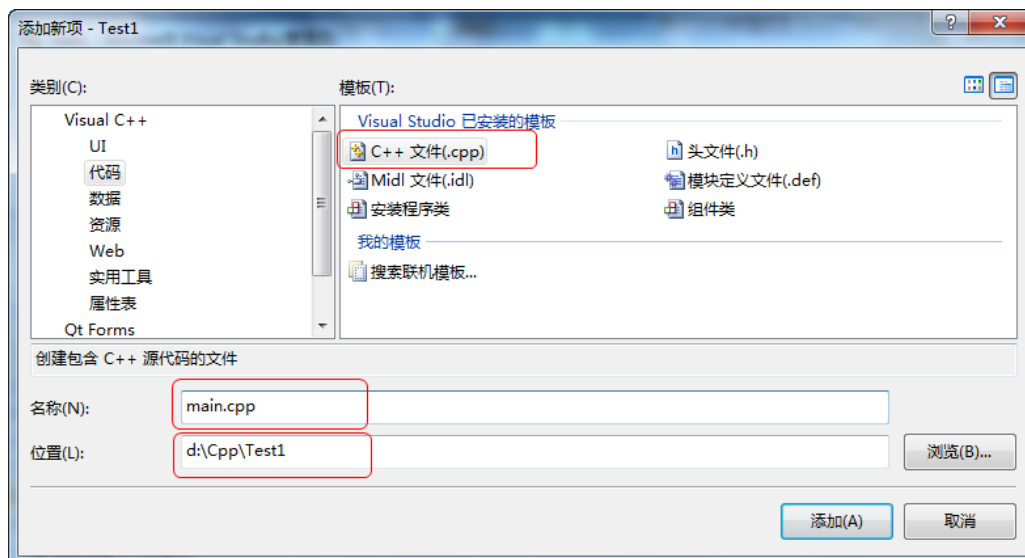
9.3.1 添加 cpp 文件

在左侧的“解决方案资源管理器”里，右键点击项目“Test1”，在快捷菜单里选择“添加|新建项”，如下图所示，



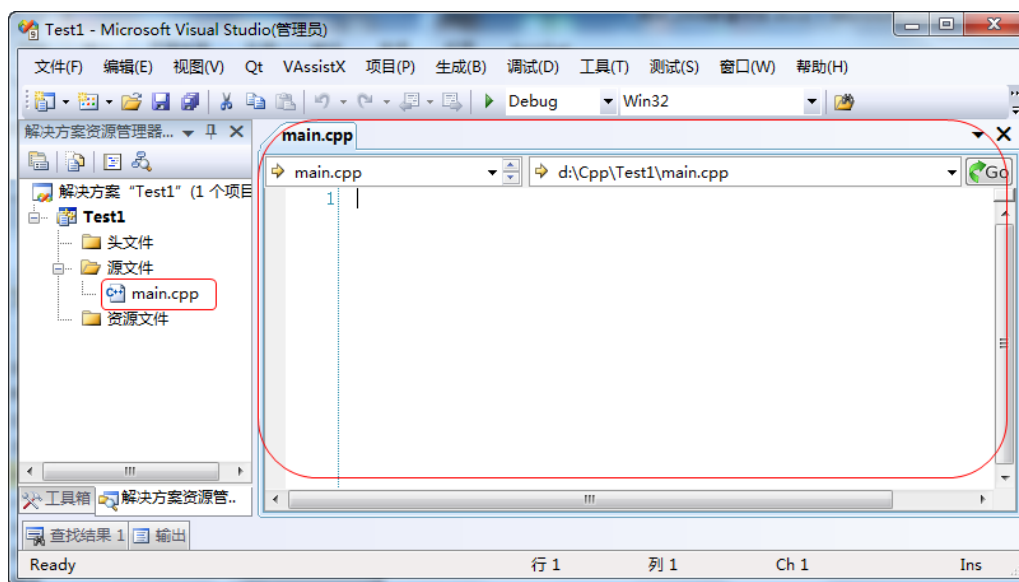
选择菜单：新建项

进入“添加新项”对话框，如下图所示，



添加新项对话框

在“添加新项”对话框里：① 左侧“类别”中选择“Visual C++|代码”，② 右上侧“模板”里选择“C++文件”，③ 下面“名称”里输入 cpp 文件的名称“main.cpp”，或者自拟其他名称，④ 下方的“位置”一栏保持不变。点“添加”按钮，则文件添加成功，如下图所示，



成功添加 cpp 文件

在主界面里，左侧文件树中，在“源文件”下面显示了刚才加入的 `main.cpp`，右侧已经打开了这个文件。

在里面输入 Hello World 的测试代码：

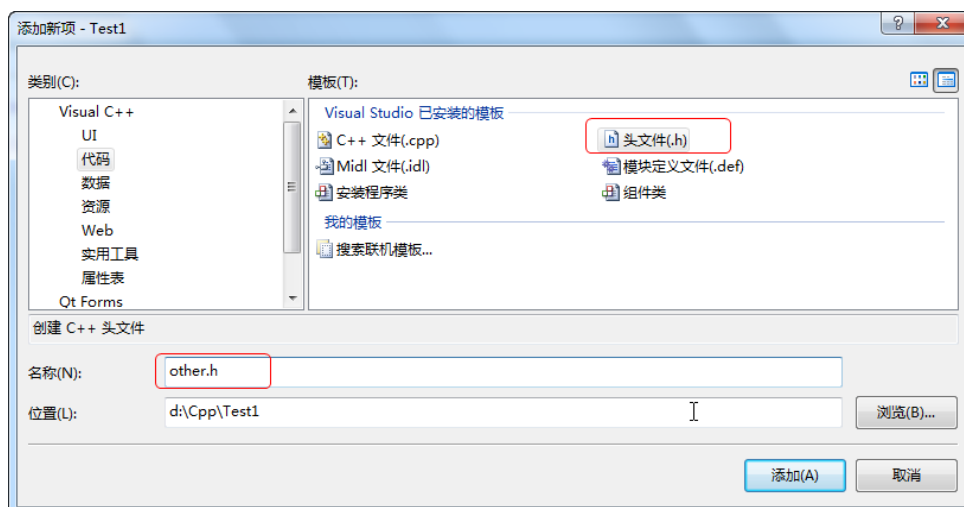
```
#include <stdio.h>

int main()
{
    printf("Hello,World!\n");
    return 0;
}
```

按 CTRL+S 保存文件，或在工具栏里选择适当的按钮来保存。如果您只需要添加一个 cpp 文件，则可以跳到本篇的第 5 节“编译和运行项目”继续后面的步骤。如果您需要往项目中添加多个 cpp 和 h 文件，则继续本节。

9.3.2 添加头文件

在左侧的“解决方案资源管理器”里，右键点击项目“Test1”，在快捷菜单里选择“添加 | 新建项”，弹出“添加新项”对话框。如下图所示，

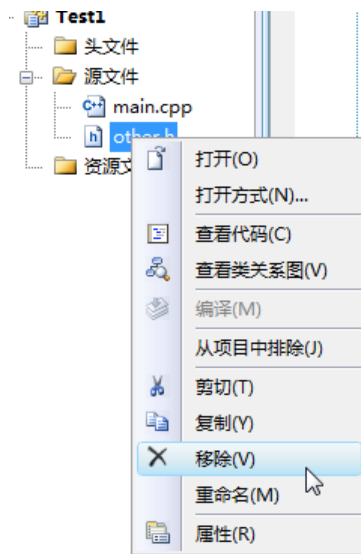


添加新项对话框

和添加 cpp 文件是类似的，只是需要 ① 将“模板”选择为“头文件”，② 输入文件名 `other.h`，注意后缀是 `h`。点“添加”按钮，即完成头文件的添加。

9.4 删除项目中的文件

在“解决方案资源管理器”中，选择要删除的 `cpp` 文件或 `h` 文件，右键点击，在快捷菜单中选择“移除”，如下图所示，



从项目中移除文件

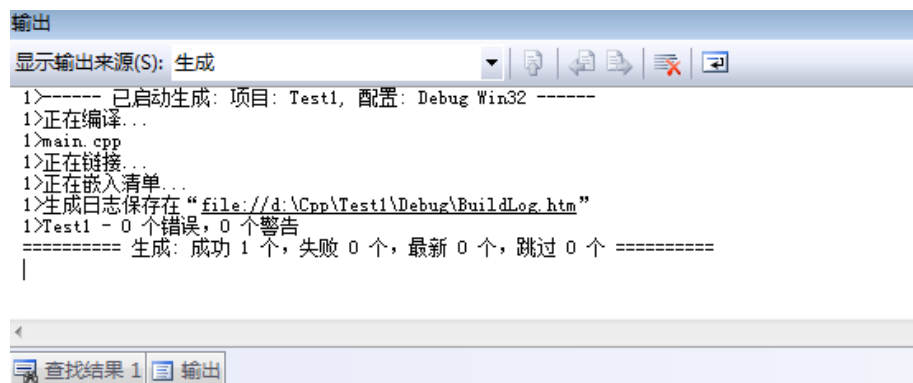
Visual Studio 2008 会弹出一个“确认对话框”，点“移除”则从项目中“移除”文件（但该文件仍然存在项目文件夹下），或点“删除”则移除该文件并将文件从项目文件夹里删除。



确认对话框

9.5 编译项目

在 Visual Studio 2008 里，代码编写完毕之后，即可以编译项目。按 F7 键开始编译，或者选择主菜单“生成|生成解决方案”开始编译。在下方的“输出”窗口会显示编译信息，如果所代码都没有语法问题，则最终显示“生成：成功 1 个，失败 0 个”。下图显示的是编译成功的输出显示，



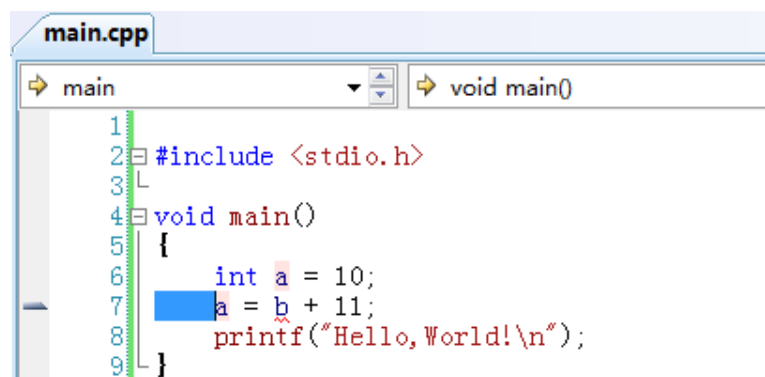
编译过程中的输出显示

如果代码中有语法错误，则编译失败，并且在“输出”窗口中给出提示：发生错误的代码位置，及可能的错误原因。如下图中，main.cpp(7) 表示其第 7 行有错，错误类型为 error C2065，错误描述为“未声明的标识符”。



编译过程中的错误显示

在“输出”窗口中双击这一行，定位到出错位置，此时程序员要仔细检查出错的位置，修正错误。然后按 F7 重新编译，直到没有任何错误提示。如下图所示，



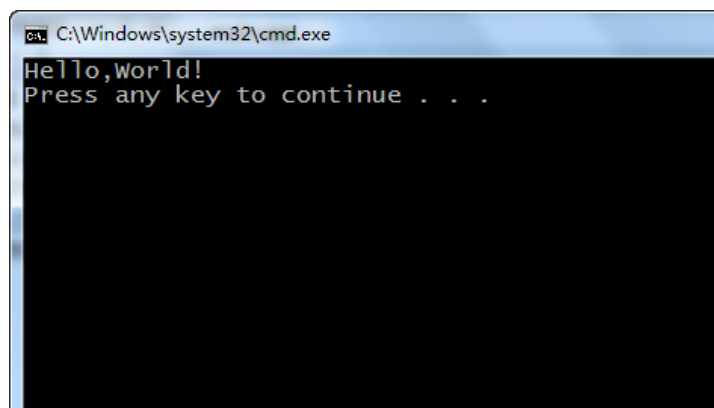
错误位置的定位

9.6 运行程序

9.6.1 在 Visual Studio 2008 中运行程序

若编译成功，所有代码都没有语法错误时，此时可以按 **CTRL + F5** 来运行程序，则可以在主菜单中选择“调试 | 开始执行”来运行程序。

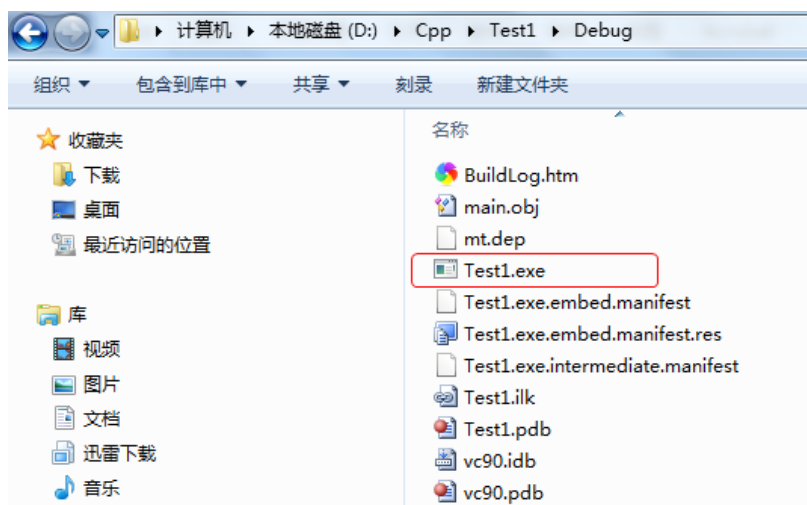
此时会弹出控制台窗口，如下图所示，



控制台窗口

9.6.2 在项目文件夹里运行程序

编译成功时，将在项目文件夹里生成一个*.exe的可执行文件。打开项目文件夹 **D:\Cpp\Test1**，打开里面的 **Debug** 子文件夹，找到后缀为 **exe** 的程序文件，如下图所示，



程序文件的位置

其中 Test1.exe 就是编译项目后得到的程序文件，双击这个文件，即可以运行之。

9.6.3 以命令行方式运行

所有的控制台程序都可以用“命令行方式”运行，具体可以参考第 18 章《多文件项目及编译过程》之“main 函数”一节。

使用命令行方式，可以指定 main 函数所需要的参数列表。下面介绍命令行方式运行程序的步骤。

首先，从 Windows 菜单里，点“运行”，输入“cmd”则打开命令行窗口（或称 DOS 命令行窗口），如下图所示，

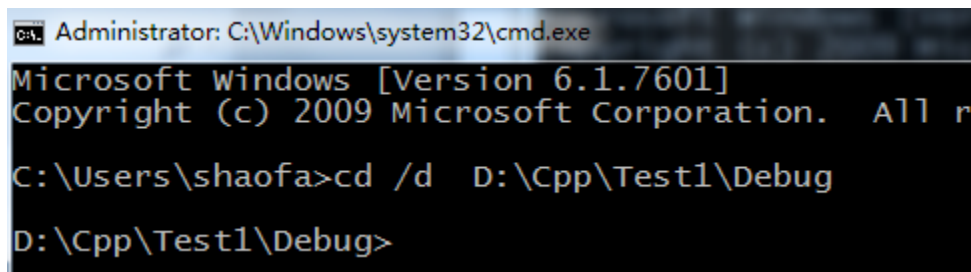


Windows7 下启动命令行窗口



Windows XP 下启动命令行窗口

进入命令行窗口，在此窗口中可以输入任何 DOS 命令。此时，我们应该先切换到程序文件的所在目录，输入 “ cd /d D:\Cpp\Test1\Debug ”，注意不要漏掉/d 参数。如下图所示，

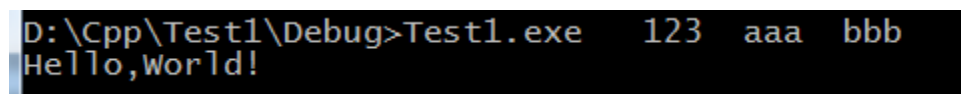


```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\shaofa>cd /d D:\Cpp\Test1\Debug
D:\Cpp\Test1\Debug>
```

切换到 Debug 目录

此时手工输入命令即可，可以 Test1.exe 后面附加输入额外的参数，这些参数将被传给 main 函数，如下图所示，



```
D:\Cpp\Test1\Debug>Test1.exe 123 aaa bbb
Hello,world!
```

输入命令行